

Linux für alle? Zur Rolle von Laien in Communities der quelloffenen Softwareproduktion

Guagnin, Daniel

Veröffentlichungsversion / Published Version

Dissertation / phd thesis

Empfohlene Zitierung / Suggested Citation:

Guagnin, D. (2020). *Linux für alle? Zur Rolle von Laien in Communities der quelloffenen Softwareproduktion*. (E-Collaboration). Glückstadt: Verlag Werner Hülsbusch. <https://nbn-resolving.org/urn:nbn:de:0168-ssoar-79794-7>

Nutzungsbedingungen:

Dieser Text wird unter einer CC BY-SA Lizenz (Namensnennung-Weitergabe unter gleichen Bedingungen) zur Verfügung gestellt. Nähere Auskünfte zu den CC-Lizenzen finden Sie hier: <https://creativecommons.org/licenses/by-sa/4.0/deed.de>

Terms of use:

This document is made available under a CC BY-SA Licence (Attribution-ShareAlike). For more information see: <https://creativecommons.org/licenses/by-sa/4.0>

Guagnin · Linux für alle?

Daniel Guagnin

Linux für alle?

**Zur Rolle von Laien in Communities
der quelloffenen Softwareproduktion**

vwh

Verlag Werner Hülsbusch
Fachverlag für Medientechnik und -wirtschaft

D. Guagnin: Linux für alle?

Bibliografische Information der Deutschen Nationalbibliothek

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet unter <http://d-nb.de> abrufbar.

© Verlag Werner Hülsbusch, Glückstadt 2020

vwh Verlag Werner Hülsbusch
Fachverlag für Medientechnik und -wirtschaft

www.vwh-verlag.de

Einfache Nutzungsrechte liegen beim Verlag Werner Hülsbusch, Glückstadt.
Eine weitere Verwertung im Sinne des Urheberrechtsgesetzes ist nur mit
Zustimmung des Autors möglich.

Markenerklärung: Die in diesem Werk wiedergegebenen Gebrauchsnamen, Handelsnamen, Warenzeichen usw. können auch ohne besondere Kennzeichnung geschützte Marken sein und als solche den gesetzlichen Bestimmungen unterliegen.

Zugleich: Diss., Technische Universität Berlin, 2019

Korrektur und Satz: Werner Hülsbusch

Umschlag: design of media, Lüchow

Druck und Bindung: Schaltungsdienst Lange oHG, Berlin

Printed in Germany

ISBN: 978-3-86488-154-1

Inhaltsverzeichnis

	Einleitung	9
	Stand der Forschung	13
	Open Source Software, ein viel beforschtes Feld	13
	Soziologische Analysen des Feldes	16
	Präzisierung der Fragestellung	23
	Struktur der Arbeit	24
	Teil I	
	Software: Entstehung und Problematisierung	
1	Eine kleine Geschichte der Software	29
1.1	Anfänge von Computer und Software	29
1.2	Kommodifizierung von Software und der Reflex der Community	34
1.3	Popularisierung und Expansion offener Software-Produktion	35
1.4	Legitimationsschwierigkeiten der Expert*innen	39
2	Implikationen von Technik und Software	43
2.1	Techno-Regulation und die Rolle von Software	44
2.2	Software wird kritisch: Critical Algorithm Studies	49
2.3	Surveillance Studies und die Rolle von Algorithmen	51
3	Software als Element moderner Technologien	55
3.1	„Jenseits des mechanischen Bewirkens“	55
3.2	Spezifische Merkmale von Software	58

Teil II Epistemische Regime zur Regulierung der Wissensdifferenz

4	Technik als soziales Verhältnis zwischen Expert*innen und Laien	69
4.1	Technik als stabiler Wirkungszusammenhang	63
4.2	Vertrauen in Technik	64
4.3	Die Macht der Expert*innen	69
5	Differenzierte Expertisegrade und die Erweiterung der Partizipation	75
5.1	Abstufungen einer „Specialist Expertise“	76
5.2	Das Selektionsproblem erweiterter Partizipation	80
6	Die Regulierung der Wissensproduktion durch epistemische Regime	85
6.1	Epistemische Regime: Vorarbeiten und Konzeption des Begriffs	86
6.2	Die soziale Ordnung der Wissensproduktion	89
6.3	Dimensionen epistemischer Regime	91
6.4	Operationalisierung zur Analyse epistemischer Regime der Softwareproduktion	96

Teil III GNU/Linux Communities als epistemische Regime

7	Untersuchungsdesign	107
7.1	Distributionen als Querschnitt verschiedener „Programmierkulturen“	103
7.1.1	„Programmierkulturen“	103
7.1.2	Charakterisierung von Distributionen	105
7.1.3	Auswahl und Beschreibung der untersuchten Distributionen	108
7.2	Sampling	112

7.3	Auswertungsmethode	115
8	Epistemische Regime der gemeinschaftlichen Softwareproduktion	119
8.1	Mitgliedschaft als kollektives Selbstbild	121
8.1.1	Ubuntu – „Linux for Human Beings“	122
8.1.2	Debian – „The Universal Operating System“	129
8.1.3	Arch – „A Simple and Lightweight Distribution“	140
8.1.4	Vergleich der Variable „Mitgliedschaft“	148
8.2	Strukturen der Community	153
8.2.1	Ubuntu – zwischen Community und Unternehmen	155
8.2.2	Debian – Demokratie der Entwickler*innen	164
8.2.3	Arch – informelle Oligarchie	170
8.2.4	Vergleich der Variable „Strukturen und Verfahren“	175
8.3	Beiträge zum gemeinsamen Wissensprodukt	184
8.3.1	Ubuntu – „Whatever skills“: Bugs und Brötchen	186
8.3.2	Debian – Developer unter sich	193
8.3.3	Arch – Userdeveloper und Developeruser	199
8.3.4	Vergleich der Variable „Beiträge“	203
9	Normative Inskriptionen in Software	211
9.1	Ubuntu Linux: In wenigen Schritten zum Betriebssystem	214
9.1.1	Download: „Help shape the future of Ubuntu“	215
9.1.2	Ubuntu: Ausprobieren und installieren	217
9.1.3	Inskription des Ubuntu-Installers	224
9.2	Debian Linux: Zwischen Anfänger- und Expertenoptionen	225
9.2.1	How to get the Debian installer	226
9.2.2	Debian Installer	227
9.2.3	Inskriptionen des Debian-Installers	236
9.3	Arch Linux: Volle Kontrolle – do it yourself	237
9.3.1	Getting and Installing Arch	238
9.3.2	Being the Install-Script	239
9.3.3	Inskription des Arch-Installers	246
9.4	Zusammenfassende Gegenüberstellung der Skripte	247

Teil IV Normative Konfigurationen

10	Zusammenfassung der Ergebnisse	261
10.1	Spezifische (normative) epistemische Regime	255
10.1.1	Usability Regime: Institutionalisierte Inklusion	257
10.1.2	Stability Regime: Demokratie qualifizierter Entwickler*innen	260
10.1.3	Minimales Eliten-Regime	264
10.2	Das Zusammenspiel der Variablen epistemischer Regime	267
10.2.1	Mitgliedschaft ist Selektionsmechanismus und Identitätsmoment	268
10.2.2	Strukturen und Verfahren als formale zentrale Governance-Struktur	270
10.2.3	Die Selektion der Beiträge strukturiert das Wissensprodukt	271
10.2.4	Normative Prägungen der Regime	273
10.3	Software: Das Wissensprodukt als Spiegel und Selektionsmechanismus der Community	274
11	Epistemische Regime verschiedener Experten-Laien-Differenzen	281
11.1	Epistemische Regime als stabile normative Ordnungen	281
11.2	Die epistemischen Rekonfigurationen der Experten-Laien-Differenz	285
11.2.1	Differenzierungen zwischen den Entwickler*innen	287
11.2.2	Beiträge jenseits einer „Contributory Expertise“	290
11.2.3	Differenzierungen von Wissen jenseits der „Interactional Expertise“	292
11.3	Spielräume zwischen Experten-Entwickler und Laien-User	293
11.3.1	Entwickler-User-Differenz	294
11.3.2	Experten-Laien-Differenz	297
12	Schlussfolgerungen	303
	Danksagung	307
	Literatur	311

Einleitung

Software gewinnt in letzter Zeit eine zunehmende Bedeutung in der gesellschaftlichen Wahrnehmung. Algorithmen und Daten fungieren als Dreh- und Angelpunkte der Verheißungen zukünftiger Innovationen, die aktuelle und künftige Probleme effizient lösen sollen. Andererseits werden sie problematisiert als undurchsichtige Kontrollarchitekturen, die selbst schwer kontrollierbare Black Boxes sind und somit latent die Demokratie gefährden.

Generell ist Technik meistens verkapselt und undurchsichtig für die User. Aus technikphilosophischer Sicht ist dies durchaus funktional. Eine arbeitsteilige Trennung von Expert*innen¹, die komplexe technische Abläufe verstehen und umsetzen, und Laien, die diese Abläufe durch Technik einfach nutzen können, erspart Letzteren eine Auseinandersetzung mit Technik, während sie aber von der Nutzung profitieren. Auf der laienhaften Nutzung von in Technik materialisiertem Expertenwissen beruhen eine Entlastung der User von Komplexität und dadurch eine Leistungssteigerung der Gesellschaft. Umgekehrt stellt sich die Frage, ob Technik, die diese Entlastungsfunktion nicht bereitstellt, also das Verständnis komplexer Zusammenhänge für die Nutzung erfordert, überhaupt noch als Sachtechnik im soziologischen Sinne gelten kann oder eher als Bastelei zu betrachten ist?

Infolge einer konsequenten Trennung von Herstellung und Nutzung der Technik sind die User darauf angewiesen, der Technik, und damit ihren Hersteller*innen und den von ihnen zugrunde gelegten Prämissen, in der Nutzung zu vertrauen, da diese allein die Funktion verstehen und kontrollieren. Dieses Vertrauen wird offenbar problematisch vor dem Hintergrund der Verbreitung von Softwaresystemen, an die einerseits soziale Bewertungen und Entscheidungsprozesse ausgelagert werden und die andererseits in ganz alltäglichen Geräten eine zentrale Rolle spielen. Das Vertrauen in einen geschlossenen Kreis von Expert*innen als alleinigen Kontrolleur*innen der Software und Inhaber*innen exklusiver Rechte, diese zu gestalten, wurde schon sehr früh von Befürworter*innen der Freie-Software-Bewegung infra-

¹ Anstelle des generischen Maskulinums wird zwecks Berücksichtigung der Geschlechtervielfalt weitgehend ein *innen („Gendersternchen“) angehängt. Ausnahmen bilden beispielsweise die Verwendung in Tabellen (aus Platzgründen), englische Wörter wie User und Developer oder zusammengesetzte Wörter.

ge gestellt. Zunächst waren dies vornehmlich Expert*innen, die es gewohnt waren, die von ihnen genutzte Software zu modifizieren. Dennoch wurde von Anfang an der Anspruch formuliert, dass *alle*, die möchten, die Möglichkeit haben sollten, die Software selbst zu verändern und die Funktionsweise einzusehen. Dafür wurden Copyright-Lizenzen entwickelt, die die Offenheit des Softwarecodes festschreiben.

Während eine anfängliche Gestaltungsoffenheit neuer Technologien häufiger zu beobachten ist, hat die Freie-Software-Bewegung Offenheit als Prinzip institutionalisiert, und zwar durch die Schaffung freier Softwarelizenzen. In der Folge hat sich der Entwicklungs-Modus von Freier/Libre Open Source Software (FLOSS)² als kollaborativer Arbeitsprozess etabliert. FLOSS hat auch in der Anwendung eine zunehmende Verbreitung erfahren, bis hinein in die Kreise von technikfernen Nutzer*innen. Beispielsweise ist Mozilla Firefox heute ein bedeutender Browser und auch Google Chrome basiert auf einem FLOSS-Projekt namens Chromium. Das Betriebssystem Linux hat sich für Rechenzentren und Webserver schon lange etabliert, mittlerweile sind auch Personal Computer mit Linux-Systemen im Handel erhältlich. Im Bereich der Kryptografie gilt FLOSS vielen als einzig vertrauenswürdiger Entwicklungsmodus, da so die korrekte Funktionsweise von verschiedenen Leuten überprüft werden kann.

Vor dem Hintergrund der Öffnungsbestrebungen der Freie-Software-Bewegung stellt sich aber die Frage, für wen die Öffnung der Software letztlich gilt. Linux ist ein Betriebssystem, das dem User vor dem Bildschirm erst den Computer als Werkzeug zugänglich macht, aber profitieren wirklich alle Linux-User von der Möglichkeit „mitzumachen“? Ist für normale Nutzer*innen am Ende nicht lediglich die *kostenfreie* Nutzung von Bedeutung? Oder spielen Laien-Nutzer*innen im Produktionsprozess doch eine aktive Rolle? Was bleibt in der Praxis vom allgemeinen Anspruch der Freie-Software-Bewegung, dass alle mitmachen dürfen? Wird die klassische Trennung von Expert*innen und Laien aufgehoben? Gewinnen Laien durch Einblicke in die Technik wirklich Vertrauen und ändern sie diese nach ihren eigenen Bedürf-

² Im vorliegenden Text wird im Folgenden der Begriff Free/Libre Open Source Software verwendet. „Free/Libre/Freie Software“ beschreibt üblicherweise eher die Ideologie und „Open Source Software“ eher den quelloffenen Entwicklungsmodus; die Trennlinie ist aber nicht immer scharf zu ziehen. Libre dient hierbei als Zusatz, um den Freiheitsanspruch gegenüber reiner Kostenfreiheit zu unterstreichen. FLOSS ist hierbei als Überbegriff zu verstehen.

nissen? Wenngleich es zunächst abwegig erscheint, dass einfache Nutzer*innen selbst Softwarecode verändern, ist es doch lohnenswert, die Rolle von Laien innerhalb verschiedener Communities genauer zu betrachten und zu fragen, in welchem Verhältnis sie zu den Expert*innen der Software-Entwicklung stehen.

Dabei lässt sich beobachten, wie sich in der Praxis der FLOSS-Entwicklung durch die Notwendigkeit einer produktiven Schließung auch hier Trennungslinien zwischen Entwickler*innen und Nutzer*innen ergeben. Aufgrund der strukturellen Offenheit des Programmcodes und der öffentlichen Zusammenarbeit benötigen diese Trennlinien eine gewisse Legitimation und sind daher Verhandlungsgegenstand der verschiedenen Communities. Daraus ergeben sich dort spannende Varianzen in ihren Praktiken, Strukturen und Normen. Diese Varianzen lassen sich fassen mit dem Begriff epistemischer Regime, der das Zusammenspiel spezifischer Praktiken, Strukturen und Normen und die Auswirkungen auf die Gestaltung gemeinschaftlich produzierten Wissens beschreibt. Innerhalb der epistemischen Regime wird ausgehandelt, wer an der Wissensproduktion teilnehmen darf und welche Beiträge anerkannt werden. Die Kriterien für die Mitgliedschaft in der Community und die Bewertung der Beiträge variieren hierbei und führen dazu, dass die Trennlinien zwischen Entwicklung und Nutzung unabhängig von Expertise variiert. Die Dichotomie von Expert*innen und Laien wird also aufgeweicht. Das Verständnis von „Expertise“ variiert zwischen den Communities und hat unterschiedliche Bedeutungen für die Autorisierung der Mitarbeit in der Community. Schließlich ergeben sich verschiedene Abstufungen von Expertise.

Für eine genauere Betrachtung der verschiedenen Abstufungen von Expertise und der Ausgestaltung des Verhältnisses von Expert*innen und Laien in FLOSS-Communities werde ich daher verschiedene Communities vergleichen. Dabei untersuche ich, welche Rolle Expertise spielt, welche Abstufungen sich innerhalb der Community ergeben und wie sich dies im Laufe der gemeinschaftlichen Zusammenarbeit auf die Wissensprodukte spezifischer Communities auswirken.

In einer vergleichenden Analyse von GNU/Linux-Distributionen gehe ich daher der Frage nach, *wie die strukturelle Beschaffenheit der Community-Organisation Normen, Praktiken und Regeln festschreibt, die als epistemische Regime die Wissensproduktion regulieren. Anschließend zeige ich exemplarisch anhand eines Software-Programms, wie sich diese Regime auf das gemeinsame Produkt auswirken.*

Es zeigt sich, dass sich die Trennlinie zwischen Expert*innen und Laien unterschiedlich ausdifferenziert in Abhängigkeit von der spezifischen Konfiguration des epistemischen Regimes. Die geteilten Normen innerhalb der Community beeinflussen die Organisationsstruktur und schließlich auch die Bewertung von Beiträgen für das gemeinsame Wissensprodukt. Folglich beeinflusst die Verfassung der Community auch das Produkt. Die Gestaltung der Software und der gemeinsamen Wissensbestände wirken sich aus auf die Zusammensetzung der Community. In einem Fall finden sich fast ausschließlich Expert*innen, darunter viele Programmierer*innen (Arch Linux), in einem anderen Fall finden sich auch weniger technisch versierte User (Ubuntu Linux). Im dritten Fall spielen Administrator*innen, die zwar nicht selbst programmieren, aber sich technisch doch sehr gut auskennen (Debian Linux), eine wichtige Rolle.

Aus den erarbeiteten Erkenntnissen lassen sich Schlüsse ziehen über die soziale Bedingtheit der Wissensproduktion. Eine wesentliche Rolle spielt demnach der normative Ausgangspunkt der Communities, da dadurch die Organisation und die Bewertungskriterien beeinflusst werden. Im Zusammenspiel von Organisationsstruktur, Entwicklungspraktiken und normativer Ordnung ergibt sich ein stabiles Regime, das die Art der Wissensprodukte prägt. Die Experten-Laien-Differenz wird dabei aufgebrochen in verschiedene Stufen von Expertise, die aber je nach Community unterschiedliche Relevanzen aufweisen und verschiedene Beteiligungsmöglichkeiten der Community-Mitglieder definieren. Tatsächlich sind aber teilweise Beteiligungsformen von Nicht-Expert*innen zu beobachten, die zwar nicht direkt am Softwarecode mitarbeiten, aber deren Perspektive teils auf andere Weise integriert wird. In einem Fall fallen Nicht-Expert*innen allerdings aus der Community.

Die Beteiligungsmöglichkeiten variieren mit unterschiedlichen Verständnissen dessen, was als Beitrag gewertet wird und welche Art von Wissen als beitragsfähig erachtet wird. Diese normativen Setzungen fließen durch die gemeinsame Entwicklung wiederum in das Wissensprodukt ein. Schließlich regulieren die beobachteten epistemischen Regime die Legitimation der Expert*innen und das Vertrauen in die Entwicklung.

Stand der Forschung

Bevor ich mich der Ausarbeitung der Fragestellung widme, möchte ich noch einen kleinen Überblick über ausgewählte Studien und Arbeiten zu FLOSS geben, um eine grobe Skizze des fragmentierten Feldes zu zeichnen. Die Forschung zu FLOSS bildet dabei einen großen Korpus, der sich aus unterschiedlichsten Disziplinen speist.

Open Source Software, ein viel beforschtes Feld

Wenngleich es einen riesigen Korpus an Studien zu FLOSS gibt, setzen sich die wenigsten kritisch mit den normativen Setzungen der FLOSS auseinander oder arbeiten gar die Wirkweise der spezifischen sozialen Ordnungen der Communities heraus. Dennoch will ich das Feld der Literatur grob abstecken und auf einige Erkenntnisse der Studien eingehen. Insbesondere zu Aspekten der ökonomischen Innovation und wettbewerblichen Dynamiken sowie der Motivation der Entwickler*innen und der Organisation von Open-Source-Projekten wurde zu Beginn des Jahrtausends reichlich Forschung betrieben.

Beispielhaft sei hier verwiesen auf eine Reihe von Studien zum ökonomischen Einfluss von FLOSS auf Innovation und Wettbewerb im Informations- und Kommunikations-Sektor (vgl. z.B. Economides/Katsamakos 2006; Ghosh u.a. 2006) und einen zusammenfassenden Überblicksartikel über Forschung zu Motivationen, Governance, Organisations- und Innovationsprozessen und wettbewerblichen Dynamiken von Krogh und Hippel (2006).

Wichtiger für die Frage nach den Verhältnissen verschiedener Arten von Mitgliedern innerhalb der Community sind aber Studien zur Rolle von Wissen und sozialen Ordnungen innerhalb der Communities. Diese bleiben aber häufig bei der Feststellung von statistischen Zusammenhängen oder beziehen ihre Folgerungen auf das ökonomische und innovative Potenzial. Beispielsweise arbeitet Kuk (2006) anhand einer quantitativen Analyse der KDE-Mailingliste heraus, dass die aktive Teilnahme sehr ungleich verteilt ist und sich vor allem durch Interaktionen qualifizierter Entwickler*innen auszeichnet. Er bezieht seine Folgerungen vor allem auf die *Effizienz* der richtigen Balance der Varianz zwischen kompetenteren und weniger kompetenten Community-Mitgliedern. Wie aber Kompetenz hier verhandelt wird und wie sich die Interaktionen gestalten, bleibt außen vor.

Andere Studien untersuchen die Interaktionsmuster in Open Source Communities. Sie untersuchen über eine Kombination von Netzwerkanalysen über Mailinglisten und der Betrachtung ausgewählter Kommunikationsverläufe (vgl. Sack u.a. 2006) die Bedeutung sozialer Beziehungen für die erfolgreiche Platzierung technischer Beiträge (vgl. Ducheneaut 2005). Ferner werden Zusammenhänge zwischen den Rollen der Teilnehmer*innen und den Interaktionsverläufen in Mailinglisten (vgl. Barcellini u.a. 2005) betrachtet und *Cross-Participants* identifiziert, Teilnehmer*innen verschiedener Mailinglisten, die als *Boundary Spanner* (vgl. Barcellini/Détienne/Burkhardt 2009) beziehungsweise *Knowledge Broker* (vgl. Sowe/Stamelos/Angelis 2006) eine wichtige Funktion erfüllen, indem sie Wissen zwischen Mailinglisten übertragen. Eine strukturelle Betrachtung der sozialen Organisation der Community wird hierbei aber nicht geleistet.

Andere Studien belegen, dass die Motivation für aktive Partizipation sowohl in Linux User Groups (vgl. Bagozzi/Dholakia 2006) als auch in der Open-Source-Software-Entwicklung (vgl. Shah 2006) oft konkrete Probleme darstellen, aber die Aktiven durch das Engagement eine Verbindlichkeit entwickeln und anschließend über längere Zeiträume hinweg dabei bleiben. Offensichtlich spielen häufig auch die politischen Ziele der Freie-Software-Bewegung eine wichtige Rolle als Motivation für aktive Teilnehmer*innen in den Communities (vgl. Lakhani/Wolf 2005).

Der Einstieg in Open Source Software Communities ist Gegenstand von jüngeren Studien. Beispielsweise untersuchen Jensen, King und Kuechler (2011) den Umgang verschiedener Communities (Wikipedia, Gimp, SVN) mit Einsteigern, beobachten unterschiedliche Antwortarten auf Einsteigerfragen und finden dabei freundlichere Antworten in Userlisten, aber in Developerlisten tendenziell hilfreichere Antworten. Auch professionelle Programmierer*innen haben offenbar Schwierigkeiten beim Einstieg in die spezifischen Gepflogenheiten und Praktiken einzelner Communities (vgl. Davidson u.a. 2014). Einstiegspfade in die Beitragskultur sind also auch mehr und mehr Gegenstand von Forschung – und auch hier kann die vorliegende Arbeit soziologische Einblicke in die kulturellen Bedingungen der Communities geben.

Allermeistens haben die Studien zu Open Source Software einen Hang, sich mit den Entwickler*innen zu beschäftigen, was auch der Tatsache geschuldet sein mag, dass jeder User potenziell auch Entwickler*in ist. Dies vernachlässigt aber die variable Beteiligung von Usern in der Produktion von Wissen – im Sinne von Software und dem Wissen über die Software in Form

von Dokumentation und Hilfestellung – sowie die Differenzierung der beteiligten User in verschiedene Grade von Expertise. Generell bleibt auch die spezifische Rolle der Organisationsstrukturen innerhalb der Communities und ihre regulierende Wirkung außer Acht oder die Betrachtung bleibt beschränkt auf einzelne Aspekte von Einzelfallbetrachtungen.

Die Rolle von Nicht-Entwickler*innen, Laien und Usern in FLOSS-Communities wurde lange vernachlässigt (vgl. Iivari 2009b). Iivari (2009a) analysiert die Bedeutung des Users in der Literatur über Open-Source-Software-Entwicklung, betrachtet die gegenseitige Konfiguration der beiden Akteursgruppen (nach Woolgar 1991; Mackay u.a. 2000) und weist dabei auf einen Bedarf an Forschung über die Rolle der User in der Praxis hin. Divitini u.a. (2003) arbeiten in ihrer Studie über die Kommunikation innerhalb von Softwareprojekten außerdem die Bedeutung politischer Aspekte von Aushandlungsprozessen heraus, in deren Rahmen Code bewertet und Design-Entscheidungen getroffen werden, und stellen auch hier Forschungsbedarf fest.

Die meisten Studien beinhalten zwar Hinweise zur Rolle von Sozialisationsprozessen (vgl. Ducheneaut 2005), Exklusionsmechanismen (vgl. Rajanen/Iivari/Lanamäki 2015) und Kommunikationsstrukturen (vgl. Barcellini/Détienne/Burkhardt 2008), bleiben aber mit quantitativen Auswertungen von Mailinglisten und Online-Repositorien (z.B. Sourceforge) für eine qualitative Betrachtung der dahinter liegenden Prozesse wenig ertragreich. Eine Reflexion über die normativen Grundlagen einer Differenzierung verschiedener Kompetenzstufen liegt meist nicht im Fokus der Betrachtung. Wie beispielsweise das Wissen durch die identifizierten Akteure nun übertragen und kommuniziert wird und welche sozialen Ordnungen die sozialen Beziehungen der Teilnehmer strukturieren, wird allenfalls am Rande diskutiert. Häufig wird hingegen ein vereinfachtes Selbstbild der Community als gegeben übernommen sowie eine Aufhebung der Experten-Laien-Differenz fraglos akzeptiert. Hier möchte ich ansetzen mit einer kritischen Betrachtung der normativen Setzungen der Communities und der spezifischen Grenzziehungen zwischen Expert*innen und Laien sowie zwischen Entwickler*innen und Usern.

Daher verfolgt die vorliegende Arbeit ein qualitatives Forschungsdesign und fokussiert die sozialen Ordnungen und Praktiken, um daraus Schlüsse zu ziehen über normative Grundierungen der epistemischen Regime und die Rolle verschiedener Grade von Expertise und schließlich deren Auswirkungen auf das produzierte Wissen zu analysieren.

Governance hybrider FLOSS-Communities

In der Forschung zu Communities gibt es einige Arbeiten, die sich mit epistemischen Grundlagen von Communities und der Governance von Communities beschäftigen. Dabei werden verschiedene Begriffe verhandelt, die im Bezug auf FLOSS-Gemeinschaften verwendet werden, von denen ich mich im Folgenden aber abgrenze, um zu begründen, warum sie für meine Perspektive nicht passend erscheinen.

O'Mahony und Ferraro (2007) haben sich mit dem Wachsen der Governancestruktur des Debian-Projektes befasst und festgestellt, dass im Zuge seines Wachstums neben dem technischen Know-how für Führungsrollen gerade Wissen über Organisationsabläufe zunehmend von Bedeutung ist. Hier schließt sich die Frage an, auf welcher Basis in Meritokratien eigentlich Verdienste und folglich Kompetenzen – im doppelten Sinne von Fähigkeit und Berechtigung für Entscheidungen – zugeschrieben werden. In einer späteren Studie konstatieren West und O'Mahony (2008) außerdem eine starke Tendenz, dass Sponsoren von Communities tatsächlich eine relativ starke Kontrolle ausüben.

West und O'Mahony (2005) problematisieren die Spannung der (finanziellen) Sponsoren in sogenannten „Sponsored Communities“ (im Gegensatz zu eigeninitiativ formierten Communities) zwischen deren Bedürfnis, die Ziele des Projekts zu steuern, und den Partizipationsmöglichkeiten von Entwickler*innen, deren freiwilliges Engagement und Commitment nicht zuletzt mit deren Gestaltungsmöglichkeiten zusammenhängt.

O'Mahony (2007) stellt weiter fest, dass mit zunehmender Diversität von finanziellen Sponsoren und Stakeholdern in Communities neue Governance-Modelle entstehen und dass es wichtig ist, gerade auch solche zunehmend „hybride“ Communities besser zu verstehen und zu untersuchen. Um ein besseres Verständnis ihrer Funktionsweise zu erlangen, kommt er zu dem Schluss, dass dafür ein Rahmen für den Vergleich von Communities notwendig ist.

In diesem Sinne werde ich verschiedene Konzepte von Kompetenz untersuchen und beleuchten, wie in verschiedenen Communities unterschiedliche Werte auch für die Gewährung von Mitspracherechten geltend gemacht werden. Dabei werde ich mich jedoch weniger auf den Aspekt der finanziellen Förderung des einen angesprochenen Fallbeispiels konzentrieren, sondern auf die normativen Ausgangspunkte der drei Vergleichsfälle fokussieren. Dafür werde ich für den Vergleich den Begriff der epistemischen Regime operatio-

nalisieren, da ich damit die relevanten Wechselwirkungen von Normen, Struktur und Praxis konzeptionell fassen kann.

Epistemic Communities und Communities of Practice

Der Begriff der *Communities of Practice* (vgl. Wenger 1998) beschreibt Problemlösungsgemeinschaften, die häufig über Firmen eingebunden sind und an gemeinsamen Gegenständen arbeiten. Dabei geht es weniger um die gemeinsame Produktion von Wissen oder eines Produktes, sondern eher um geteilte Fertigkeiten und das Wissen, um bestimmte Probleme zu lösen. Die Betrachtung von FLOSS-Communities unter der Perspektive des Begriffs *Communities of Practice* fokussiert auf den Austausch verschiedener Entwickler*innen über ihre Programmierfähigkeiten – und zwar projektübergreifend. Im Hinblick auf die gemeinsame Zusammenarbeit innerhalb einer identitätsstiftenden Community als gemeinsames Projekt, insbesondere für die Analyse der Differenzen der inneren Organisation und der spezifischen Praktiken verschiedener Communities, ist der Begriff aber wenig hilfreich.

Epistemic Communities – ein Begriff, der ebenfalls im Bezug auf FLOSS-Communities von anderen Autor*innen verwendet worden ist – bezeichnet ein politologisches Konzept, das auf die Beschreibung von Experten-Communities zielt, die sich basierend auf einem gemeinsamen, epistemischen Fachwissen gemeinsam an politischen Diskursen beteiligen.

[W]e offer an approach that examines the role that networks of knowledge-based experts – epistemic communities – play in articulating the cause-and-effect relationships of complex problems, [...] framing the issues for collective debate [...] and identifying salient points for negotiation. (Haas 1992: 2)

Der Begriff der *Epistemic Communities* (ebd.) ermöglicht sehr wohl die Betrachtung von geteilten Werten und Prinzipien, die sich von den spezifischen Praktiken innerhalb dieser Communities ableiten, welche auch im Kern der Arbeit stehen. Diese Stärke des Begriffs versuchen Edwards (2001) sowie Mateos-Garcia und Steinmueller (2008) nutzbar zu machen, um die normativen Aspekte innerhalb von FLOSS-Communities herauszuarbeiten, die in der Literatur regelmäßig vernachlässigt wurden. Allerdings liegt der Begriff einerseits auf einer anderen Ebene, nämlich der Ebene politischer Aushandlungsprozesse außerhalb der Software-Communities, und andererseits bezieht er sich explizit auf Communities anerkannter Expert*innen. Betrachten wir aber Produktionsgemeinschaften als Zusammenarbeit von Expert*innen und Laien, passt der Begriff kaum mehr. Auf einer anderen Ebene ist es sicher

sinnvoll, FLOSS-Communities als Epistemic Communities zu beschreiben, aber eben im Hinblick auf projektübergreifende geteilte Überzeugungen, die im politischen Diskurs vertreten werden: etwa die Forderung, dass Software transparent, veränderlich und unabhängig von Hersteller*innen sein muss.

Mateos-Garcia und Steinmueller (2008) kommen in ihrer Betrachtung von FLOSS-Communities als Epistemic Communities zu dem Schluss, dass das Organisations-Design der Communities zentral ist und dass nur ein Teil der Entscheidungen auf einer technischen Basis getroffen werden kann. Für deren Analyse greife ich in der vorliegenden Arbeit auf das Konzept epistemischer Regime zurück, um ganz konkret verschiedene solcher Community-Strukturen zu untersuchen und dabei auch die Frage ihrer unterschiedlichen epistemischen Auffassungen und deren Zusammenhang mit ihren spezifischen Strukturen zu beantworten.

Soziologische Analysen des Feldes

In Abgrenzung des grob skizzierten Korpus an Artikeln möchte ich nun einige detailliertere Monografien aufgreifen, die durch ihren soziologischen Blick auf das Feld und die Akteure und ihr Interesse an den qualitativen Zusammenhängen interessante Erkenntnisse für ein kritisches Verständnis des Phänomens FLOSS bergen.

Zunächst möchte ich dazu auf zwei ethnografische Studien hinweisen, die sich beide mit dem Projekt Debian auseinandergesetzt haben. „Coding Freedom“ von Coleman (2013) und „La liberté logicielle“ von Lazaro (2008) beschäftigen sich im Detail mit den Motivationen, ethischen Abwägungen der Entwickler*innen, den normativen Diskursen und der Organisation im Projekt. Diese detailreichen Einzelfallstudien dienen für die Konzeption meiner Arbeit als hilfreiche Quelle von Anregungen für interessante Aspekte bei der vergleichenden Betrachtung von GNU/Linux-Distributionen. Sie bildeten dabei eine punktuelle Ergänzung zu meiner Arbeit, die sich aufgrund des vergleichenden Charakters mit weniger Details begnügen muss und dafür mehr Wert auf eine analytische Struktur legt, um Antworten auf theoretische Fragen nach den Wirkungsweisen epistemischer Regime zu finden (s. S. 23).

Normative Bezüge und gemeinsame Strukturen der Freie/Open Source Communities

Sebald (2008) beschäftigt sich mit den Entwicklerdiskursen in Freie/Open-Source-Software-Gemeinschaften. Er arbeitet verschiedene normative Narrative heraus (Kap. 3) und untersucht die verschiedenen Kommunikations- und Produktionsstrukturen (Kap. 4). Dabei beschreibt er zunächst sehr detailliert generelle Arbeitsstrukturen und unterscheidet anschließend verschiedene Regierungsmodi unterschiedlicher Projekte. Neben Anarchie identifiziert er verschiedene Formen von Demokratie, konsensorientierte Komitees, Oligarchien, Aristokratien sowie Monokratien. Die gebildeten Typen orientieren sich dabei an den gängigen politischen Herrschaftsformen, wobei nicht „die Chance für einen Befehl Gehorsam zu finden für die Bezeichnung ausschlaggebend [ist], sondern die in der jeweiligen Herrschaftsform gegebenen Handlungs- und Teilhabemöglichkeiten der ‚Beherrschten‘“ (ebd.: 168). Sebald beschäftigt sich weniger mit der stabilisierenden Rolle von epistemischen Regimen, sondern er hebt mit seiner Untersuchung ab auf eine Analyse der Bedeutung der praktizierten Ökonomie einer offenen Wissensproduktion.

Soziale Strukturen und Mechanismen der Entscheidungsfindung

Taubert (2006) beschreibt in seiner Arbeit „Produktive Anarchie“, wie und warum sich in Freie/Open Source Software Communities bestimmte soziale Strukturen herausgebildet haben. Dabei geht er auf die Innovation freier Software-Lizenzen ein, die die legale Basis für die gemeinschaftliche Produktion schaffen, und auf die technischen Entwicklungen, die den Zugriff auf eine gemeinsame Wissens- und Produktionsbasis – insbesondere den Quellcode – ermöglichen.

Darüber hinaus untersucht Taubert an einer Einzelfallstudie des Projekts K-Mail, wie Softwareprojekte organisiert sind und wie Entscheidungen und kollektives Handeln dort trotz der freiwilligen Selbstzuordnung zum Kollektiv gemeinsam koordiniert wird. Er stellt dabei fest, dass die Kompetenz die Leistungsrollen der Entwickler*innen reguliert. Als Mechanismen der Handlungskoordination identifiziert er verschiedene Strategien. Üblicherweise werden Entscheidungen über Zielsetzungen über den Austausch von („rationalen“) Argumenten erörtert, wobei Taubert feststellt, dass einzelne Entwickler*innen durch ihre soziale Position einen besonderen Einfluss nehmen können, die allerdings in der Regel meritokratisch begründet ist, also durch lange

Zugehörigkeit oder Erfahrungen und Fähigkeiten, die in der Vergangenheit unter Beweis gestellt wurden. Taubert beobachtet, dass insbesondere im Falle von Dissens die Reputation im Feld ausgespielt wird.

Außerdem stellt er fest, dass bei uneindeutiger Sachlage eine Ermüdung der Diskussion eintreten kann und durch die Schaffung von Tatsachen durch Schreiben von Code strategisch die eigene Position untermauert werden kann.

Aufgrund der Rolle von Reputation und Kompetenz spielt seiner Argumentation zufolge auch der reine Anwender eine untergeordnete Rolle. Er hat nicht zuletzt auch aufgrund der fehlenden Möglichkeit, selbst seine Ideen umzusetzen, eine schwächere Position. Zunächst spielt die Nutzung durch Anwender*innen zwar eine Rolle für das Projekt als Feedback und Wertschätzung der Arbeit und zur Erlangung von Reputation, um wiederum auch neue Entwickler*innen zu gewinnen, aber nach Erreichen einer gewissen Etabliertheit nimmt die Bedeutung des Anerkennungsfaktors deutlich ab. Im Hinblick auf die Rolle der Anwender*innen differenziert er außerdem „Partizipation“ und „Publizität“ (ebd.: 179): Publizität bedeutet im Gegensatz zu Partizipation lediglich Transparenz der Entscheidungen und Beiträge, die reinen Anwender*innen bleiben also im Grunde auf der Ebene der Publizität.

Taubert hat in seiner Studie interessante Aspekte der Genese der sozialen Strukturen der offenen Softwareproduktion erarbeitet und damit einen wertvollen Beitrag geleistet, an dem ich anknüpfen kann. Die Bedeutung der Rolle von Kompetenz und Reputation benötigt dabei eine genauere Betrachtung, da dies sich auswirkt auf die Rolle der Anwender*innen. Die Begriffe Anwender*innen und Programmierer*innen bilden vielmehr Pole in einem Kontinuum und auch die Kompetenz differenziert sich in verschiedene Stufen aus.³ In der spezifischen Ausgestaltung des Verständnisses von Kompetenz und der anerkannten Rollen innerhalb einer Gemeinschaft differenzieren sich verschiedene Nuancen zwischen Kern- und Peripherie-Entwickler*innen und Power- und Laien-Usern aus, die in der Konstruktion der Organisationsmodelle spezifischer Communities begründet liegen.

Die generelle Betrachtung der Strukturen und Mechanismen von Innovationsnetzwerken werde ich daher um eine vergleichende Analyse spezifischer Communities ergänzen, um das Zusammenspiel von gemeinsamen Interessen, normativen Orientierungen und den strukturellen Eigenschaften verschiedener Gemeinschaften zu beleuchten.

³ um nicht zu sagen: auch in verschiedene Themenbereiche

Zwischen Utopie und Zweckmäßigkeit

Schrape (2016) untersucht die Rolle von Unternehmen in Open Source Software und kommt zu dem Schluss, dass Open Source sich zu einer Innovationsstrategie kommerzieller Unternehmen etabliert hat. Dabei zeichnet er die historische Entwicklung kollektiver Invention und Produktion nach, von der Entstehung über die Kommodifizierung bis zur Insitutionalisierung von Open Source Software und zum starken Wachstum seit der Jahrtausendwende. Dabei öffnet seine Perspektive der Entstehung von FLOSS interessante Bezugspunkte. Er konstatiert eine frühe Verzahnung von offenem Austausch und Proprietarisierung und Kommodifizierung von Inventionen. Hierbei bildet die Innovation freier Lizenzen einen Eckpunkt, der das Teilen von dem in Software materialisierten Wissen auf eine legale Basis stellt. Er teilt die Geschichte der Open Source Software in drei Abschnitte ein: Freie Software als Utopie infolge der Gründung von GNU 1983 durch Richard Stallman (s. auch Kap. 1 in dieser Arbeit), die Entwicklung von Open Source als eine Methode durch die Gründung der Open-Source-Initiative in den 1990ern, die explizit den Business-Sektor adressiert, sowie die Etablierung als Innovationsstrategie seit der 2000er-Jahre.

Er beleuchtet dabei die Rolle des Engagements von angestellten Entwickler*innen in Open-Source-Projekten und erarbeitet verschiedene Koordinationsmodi entlang der Achsen *Bedeutung des Einflusses von Unternehmen* und *Grad der hierarchischen Strukturierung* der Community-Governance und bildet vier Idealtypen: *korporativ geführte Kollaborationsprojekte*, die maßgeblich von Unternehmen angeleitet werden, *heterarchisch angelegte Infrastrukturvorhaben*, die dezentralere Entscheidungsmuster aufweisen und meist nicht durch kommerzielle Akteure initiiert werden, aber im Laufe der Zeit zunehmend von ihnen getragen werden, *elitezentrierte Projektgemeinschaften*, die einen moderaten Einfluss von Unternehmen aufweisen und durch eng definierte Kerngruppen angeführt werden, sowie *Peer Production Communities*, die marktunabhängige Kollaboration unter Gleichberechtigten darstellen.

Schrape resümiert, dass insbesondere kleine Nischenprojekte mit weniger und flacheren Strukturen auskommen, hingegen im Zuge des Wachstums des Projekts aber häufig „kathedralenartige“ Strukturen entstehen. Da die Strukturen aber auf unterschiedliche Weise und zu unterschiedlichen Graden heterarchisch oder hierarchisch angelegt sein können und auch die Einflussnahme von Unternehmen vielfältig sein kann, sind hier weitere detailliertere Analy-

sen notwendig. Zudem sind die Faktoren für die Ausbildung verschiedener Organisationsstrukturen auf verschiedene Gründe zurückzuführen, von denen die Größe der Community nur einen Faktor darstellt. Vielmehr sind diese auch auf unterschiedliche normative Vorstellungen der Communities zurückzuführen, wie ich im weiteren Verlauf herausarbeite.

Eine Einführung von hierarchischen Entscheidungsstrukturen im Laufe des Wachstums der Community hat auch die Wikipedia verzeichnet, wie Stegbauer (2009) treffend analysiert, was auch als ein Art Wandel vom egalitären Ideal zu einem pragmatischen Modus Operandi verstanden werden kann. In der Studie werden verschiedene Rollen in der Community herausgearbeitet und gezeigt, wie die Zuweisung von Rollen zusammenhängt mit dem erbrachten Engagement. Diese Erkenntnisse fügen sich ein in die hier für verschiedene Linux Communities beschriebenen Auswirkungen der unterschiedlichen Ausgestaltung von Mitglieder-Rollen und die jeweils damit verbundenen Anforderungen. Darüber hinaus vergleicht Stegbauer die Unterschiedlichkeit der Artikel der englischsprachigen und der deutschsprachigen Communities und beschreibt zumindest die unterschiedlichen Ausgestaltungen der Wissensprodukte basierend auf unterschiedlichen normativen Prämissen. Während es in der deutschsprachigen Community eine Präferenz für die Auswahl *einer* „richtigen“ Bedeutung gibt, scheint es in der englischsprachigen Community angebracht, verschiedene Meinungen nebeneinander stehen zu lassen. Eine systematische Untersuchung der strukturellen Unterschiede der epistemischen Regime unterbleibt aber bei Stegbauer.

Wie gezeigt gibt es mannigfaltige Anknüpfungspunkte für eine systematische Analyse epistemischer Regime in offenen Softwaregemeinschaften. Die vorgestellten Studien liefern eine Menge Erkenntnisse, die auch in die Überlegungen bei der Konzeptionalisierung der vorliegenden Arbeit eingeflossen sind. Was bislang fehlt und was ich im Folgenden ergänzen möchte, ist eine systematisch vergleichende Analyse epistemischer Regime und deren Auswirkungen auf die Wissensproduktion, wie ich sie im Folgenden erarbeiten werde.

Präzisierung der Fragestellung

Im Folgenden möchte ich nun die Fragestellung für die theoretische Auseinandersetzung und die empirische Analyse der Arbeit präzisieren:

Wie ich in Teil I herausarbeite, hat die Freie-Software-Bewegung über die Schaffung von sogenannten freien Lizenzen die Trennung zwischen den Hersteller*innen und Nutzer*innen im Prinzip aufgehoben. Es lässt sich nachzeichnen, wie trotz der formellen Aufhebung eine Differenz zwischen Hersteller*innen und Nutzer*innen emergiert, die verbunden ist mit Wissensunterschieden. Jedoch differenziert sich das Verhältnis von Expert*innen und Laien aus. Aus einer Dichotomie von Technik herstellenden Expert*innen und Technik nutzenden Laien entstehen graduelle Stufen von Expertise und User, die sich auf verschiedene Weise an der Produktion beteiligen.

Interessant ist hierbei, wie unterschiedliche Communities verschiedene Strukturen entwickeln, um die Wissensdifferenzen zu managen und die gemeinsame Wissensproduktion zu regulieren. Die herrschenden Normen, Regeln und Praktiken lassen sich mit dem Begriff *epistemische Regime* als regulierende Strukturen der Wissensproduktion fassen. Wissensproduktion meint hierbei die Produktion von Software als einer Form von Wissen sowie die dazugehörigen Dokumentationen.

Für die Herausarbeitung der Wirkungsweise epistemischer Regime vergleiche ich verschiedene Communities offener Softwareproduktion, um die spezifischen vorherrschenden epistemischen Regime miteinander zu vergleichen und die daraus folgenden Einflüsse auf das gemeinsam produzierte Wissen zu analysieren. Dabei unterscheiden sich die betrachteten Communities schon in ihrem Selbstverständnis, was ich als normativen Ausgangspunkt betrachte, der die emergierenden Strukturen determiniert. Infolge der wirksamen Strukturen bildet sich ein Wechselverhältnis im Sinne einer strukturierenden Struktur, die auf die Adressierung verschiedener Nutzerbilder zurückwirkt. Dabei fokussiere ich insbesondere den Einfluss der spezifischen Konfigurationen der epistemischen Regime auf die Wissensprodukte und die Reproduktion und Einschreibung der identifizierten normativen Orientierungen der Communities.

Die zentralen Fragen der Arbeit lauten daher: *Wie prägen geteilte normative Vorstellungen der Mitglieder einer Freie/Open Source Software Community spezifische epistemische Regime und wie wirken sich diese Regime aus auf gemeinsame Wissensprodukte?*

Die grundlegende normative Orientierung verorte ich dabei in der Bewertung verschiedener Formen von Expertise und Nicht-Expertise und der damit verbundenen Integration von Laien in der Partizipation innerhalb der Community.

Struktur der Arbeit

Die vorliegende Arbeit ist gegliedert in vier Teile. Im *ersten Teil* werde ich die gesellschaftliche Rolle von Software und ihre Problematisierung diskutieren und beleuchten, warum Software als spezifische Sachtechnik eine Kritik der üblichen arbeitsteiligen Trennung von Herstellung und Nutzung mit sich bringt. Dabei beginne ich bei der Etablierung von Software als eigenständigem Produkt und der damit verbundenen Kritik an ihrer Kommodifizierung und Proprietarisierung in den 80er-Jahren. Aus dieser Kritik ging die Freie-Software-Bewegung und die Institutionalisierung eines kollaborativen Entwicklungsmodus durch freie Lizenzen hervor (Kap. 1).

Unabhängig davon wird Informationstechnik und ihr besonderes Einflusspotenzial – beziehungsweise die doppelt angelegte Kontrolle im Sinne von Überwachung und Steuerung – in verschiedenen wissenschaftlichen Disziplinen diskutiert. Aus der gesellschaftlichen Kritik der *Techno-Regulation*, der *Critical Algorithm Studies* und der *Surveillance Studies* lassen sich bestimmte spezifische Merkmale von „moderner“ Technik ableiten, die sich im Wesentlichen mit besonderen Eigenschaften von Software decken (Kap. 2).

In Kapitel 3 systematisiere ich diese besonderen Eigenschaften mit Rückgriff auf Werner Rammerts (2006) Dimensionen technischen Wandels und begründe so eine Besonderheit von Software, die den Ausgangspunkt bildet für eine theoretische Diskussion der für Technik im Allgemeinen typischen Trennung von *Herstellung* durch Expert*innen einerseits und ihre *Nutzung* durch Laien andererseits.

Im *zweiten Teil* widme ich mich der theoretischen Diskussion der im ersten Teil ausgearbeiteten sozialen Kritik des Verhältnisses von Expert*innen und Laien und setze einer Experten-Laien-Dichotomie eine ausdifferenzierte Abstufung verschiedener Grade von Expertise entgegen. Für die Analyse der Selbst-Regulierung offener Produktionsgemeinschaften erarbeite ich eine Operationalisierung des Konzepts epistemischer Regime.

Ausgehend von Ingo Schulz-Schaeffers Sozialtheorie der Technik (2000) ergibt sich aus dem spezifischen Kontrollpotenzial von Software hier eine besondere Rolle von Vertrauen in die Richtigkeit der zugrunde gelegten Prinzipien und der Macht derer, die die Regeln ihres Funktionierens definieren (Kap. 4).

Eine Analogie in der Kritik von Expertise findet sich in der Dekonstruktion wissenschaftlicher Autorität in den Science Studies. Dort entwickeln Harry Collins und Robert Evans (2007) verschiedene Stufen von Expertise im Sinne einer partizipativen Erweiterung der gestaltenden Akteure. Deren Konzept übertrage ich auf die hier betrachteten offenen Produktionsgemeinschaften *technischen* Wissens (Kap. 5).

Die Öffnung der Herstellung von Technik hin zu einer offenen Produktionsgemeinschaft führt im Fall von Linux zu einer epistemischen Regulierung der gemeinsamen Produktion *technischen* Wissens. Daher erarbeite ich in Kapitel 6 einen Analyserahmen für die vergleichende Untersuchung epistemischer Regime der Softwareproduktion.

Im *dritten Teil* entwickle ich das Untersuchungsdesign eines Vergleichs dreier Linux-Communities, um die spezifischen Konfigurationen der jeweiligen epistemischen Regime empirisch zu untersuchen, und zeige anschließend deren Wirkung beispielhaft anhand des Designs eines konkreten vergleichbaren Software-Programms, des Installationskripts.

In Kapitel 7 stelle ich die Fälle kurz vor und motiviere das Sampling und die Methoden der durchgeführten Untersuchung. Anhand der in Kapitel 6 erarbeiteten theoretischen Variablen epistemischer Regime beschreibe ich die empirischen Befunde der drei untersuchten Fälle und fasse die Ergebnisse jeweils tabellarisch zusammen (Kap. 8). In einem weiteren Schritt fokussiere ich die konkreten Software-Implementierungen zur Installation des Systems. Hier lassen sich sehr gut die epistemischen Setzungen der verschiedenen Communities illustrieren. Für den Vergleich verwende ich den Analyserahmen des Berlin Script Collective (2018).

Im *vierten* und abschließenden *Teil* der Arbeit fasse ich die Ergebnisse zusammen und diskutiere die weiterführenden Erkenntnisse, die sich aus der Analyse ergeben.

In Kapitel 10 beschreibe ich fallspezifisch, wie in den vorgefundenen epistemischen Regimen die verschiedenen Variablen zusammenwirken, und betrachte anschließend das Zusammenspiel der Variablen innerhalb der Regime sowie die Wechselwirkung zwischen den Regimen und ihrer Wissensprodukte. Schließlich arbeite ich heraus, was sich daraus für Folgerungen

ergeben für die Bedeutung von Expertise und dem Verhältnis von Herstellung und Nutzung in den offenen Produktionsgemeinschaften der FLOSS-Communities (Kap. 11).

Abschließend folgen einige Schlussbemerkungen und offene Fragen für zukünftige Studien.

Teil I

Software: Entstehung und Problematisierung

1 Eine kleine Geschichte der Software

Zunächst werde ich die Geschichte der Freie-Software-Bewegung umreißen, um den Wandel von FLOSS von einer reinen Experten-Community hin zu einer breiten Mischung verschiedener Arten von Usern, die die Software zu einem guten Teil schlicht als Produkt *benutzen*, herauszuarbeiten.

Mein Fokus liegt hierbei auf der Gründungsmotivation der Freie-Software-Bewegung, die ihren Ausgangspunkt in der Frage der Gestaltungsmöglichkeiten der Software durch ihre Nutzer*innen hat. Die Frage, wer Software denn beziehen, verändern und Anderen zur Verfügung stellen kann, ist hierbei zentral für die Rolle verschiedener Arten von Mitgliedern – Expert*innen und Laien – und die empirische Betrachtung der potenziellen Aufhebung der Trennung von Herstellung und Nutzung.

Zunächst beginne ich bei der Entstehung von Computern und der anfänglichen Notwendigkeit der User, die Maschine über eigene Programmierung zu bedienen. Anschließend beschreibe ich die Kommodifizierung von Software und die damit verbundene Proprietarisierung. Diese riefen Widerstände in der frühen User-Basis hervor, die infolgedessen ihre Forderung, die Software weiterhin selbst programmieren zu dürfen, institutionalisierten. Schließlich wurde FLOSS zunehmend populär und auch für Laien relevant, was erst die der Arbeit zugrunde liegenden Differenzierungen zwischen Experten- und Laien-Nutzer*innen begründet.

Zuletzt fasse ich die Ideologie Freier Software als eine Problematisierung der Legitimation von Macht über Technik zusammen. Dabei ziehe ich eine Analogie zur Kritik wissenschaftlicher Autorität und der damit verbundenen Legitimitätskrise, wie sie von Collins und Evans (2007) diskutiert werden.

1.1 Anfänge von Computer und Software

Computer waren anfänglich, also in den frühen 70er-Jahren, große Rechenmaschinen, die im Grunde nur von relativ wenigen Spezialist*innen verwendet wurden, um damit Dinge zu machen, die im Vergleich zu Smartphone-Spielen oder den sogenannten „Apps“, die in Verbindung von Smartphones

vornehmlich Programme zur Alltagsbewältigung darstellen, recht abstrakt und komplex erscheinen. Computer waren anfänglich insbesondere *Rechenmaschinen*. Die Art, Dinge am Computer zu erledigen, war außerdem sehr entfernt von der heute etablierten grafischen Bedienung über Touch-Displays und „Mäuse“. Wer einen Computer bediente, programmierte im Grunde den Computer, das Betriebssystem stellte dafür nur grundlegende Befehle bereit und wurde häufig einfach mit dem Computer mitgeliefert, teils im Paket mit dem Quellcode und teilweise in Kombination mit Schulungen, wie diese zu benutzen sei.

Die Bedienung erfolgte direkt über eine sogenannte Kommando- oder Befehlszeile (bzw. noch früher über Lochkarten, s. S. 31), wie sie vor der Etablierung fensterbasierter grafischer Desktop-Oberflächen üblich waren, wie sie aber insbesondere im administrativen Bereich durchaus noch Verwendung finden. Im Grunde ist diese Art der Computer-Bedienung nicht allzu weit weg vom Programmieren, da in der Kommandozeile Computer-Befehle eingegeben werden, die verschiedene Parameter und Argumente enthalten und so zu bestimmten Ergebnissen und Bildschirm-Ausgaben führen. Das über den Befehl aufgerufene Computerprogramm besteht seinerseits wiederum aus Teilprogrammen, die selbst Parameter und Argumente weiterreichen, um kleine Aufgaben zu erledigen, und die Ergebniswerte für die Weiterverwendung als Parameter zurückgeben. Dabei greifen die Programme wiederum auf vordefinierte Befehls-Bibliotheken der Programmiersprache oder des verwendeten Betriebssystems zurück, welches letztlich Befehle bereitstellt, die die einzelnen Hardware-Komponenten ansteuern.

Es handelt sich also um verschiedene Ebenen von Abstraktionsleveln, wobei die User auf verschiedenen Ebenen modifizierend eingreifen können, insofern die tieferen Ebenen für eine Modifikation zugreifbar sind. Im Hinblick der Kommandozeile und vor dem Hintergrund der Gewohnheit, mit Befehlen und Textausgaben zu hantieren, erscheint es folglich naheliegend, dass User bei auftretenden Problemen das Bedürfnis verspüren, eine Ebene tiefer zu gehen, um die nächste Ebene von Befehlen zu überprüfen und zu schauen, ob dort das Problem eventuell behoben werden kann oder vielleicht eine nicht vorhandene Funktion eingefügt werden kann. Insbesondere vor dem Hintergrund, dass in den frühen Tagen der Computer die Software noch nicht in dem Maße ausgereift war, wie das heute der Fall ist, gab es weniger im Betriebssystem enthaltene Funktionen, die daher vom User implementiert werden mussten, und es gab mehr Fehler, die gefunden und behoben werden wollten (vgl. bspw. Grassmuck 2004: 202, 210). User stießen also häufiger

auf die Grenzen der (üblicherweise mitgelieferten) Programmierung ihres Computers und ergänzten ihre eigenen Erweiterungen zum bestehenden „Operating System“ hinzu, was somit zur der alltäglichen Praxis der Computer-Nutzung gehörte.

„Prähistorischer“ Exkurs

In diesem Zusammenhang soll kurz Erwähnung finden, dass die Kommandozeile auch schon ein großer Entwicklungsschritt war gegenüber den noch früheren Generationen von Rechnern, die noch per Lochkarte programmiert wurden oder bei denen die Programmierung durch Draht-Verbindungen hergestellt wurde. In diesen „prähistorischen“ Tagen wurden Programmierbefehle noch von Mensch zu Mensch erteilt. Mathematiker riefen ihren Assistentinnen (den „Computer Girls“) zu, welche Drähte nun verschaltet werden sollten (vgl. Chun 2011: 29 ff.). Über die Programmierung von Lochkarten bis hin zur Kommandozeile wurde die Programmierung also gewissermaßen „weicher“ oder dematerialisiert: weg von Draht und Lochpapier hin zur visuellen Repräsentation durch flüchtig und veränderbar bildlich dargestellte Zeichen. Ein wichtiges Detail der verbalen Computerbefehle ist dabei das Erscheinen einer weiteren Abstraktionsebene: Von den logischen Schaltungen, die letzten Endes Informationen über die Unterscheidung von Stromstärken⁴ verarbeiten, wird abstrahiert zu Hexadezimalzahlen⁵ beziehungsweise binären Zahlen, die in ihrer konkreten elektronischen Entsprechung Null und Eins oder Strom und Nicht-Strom symbolisieren. Diese Zahlen repräsentieren sowohl Daten als auch Befehle, die in logischen Schaltungen (Hardware) materialisiert sind, wobei die Befehle auf die Daten angewendet werden. Für sogenannte *Hochsprachen*, die von den technischen Zusammenhängen abstrahieren, werden diese Zahlen und Maschinenbefehle übersetzt in verbalsprachliche Funktionsbefehle wie „print“ (Ausgabe), „do loop“ (Schleifen), „return“ (Rückgabewerte) oder „exit“ (Programmende) und Bedingungen wie „if, then, else“. Diese, für Menschen deutlich leichter nach-

4 Verschiedene Varianten davon sind Draht oder kein Draht bzw. Licht oder kein Licht (Lochkarten) oder Strom oder nicht Strom – bzw. zwischen 0 und 5 V wechselnde Strom-Spannungen, die in bestimmten Bereichen als 0 oder eben 1 interpretiert werden.

5 Zur übersichtlicheren Darstellung wird Binärcode üblicherweise im 16er-System dargestellt. Vier Bits, die jeweils den Wert 0 oder 1 annehmen können und somit in ihrer Kombination 16 (also 2⁴) verschiedene Werte repräsentieren (0–15 im Dezimalsystem), werden hierbei in einer Hexadezimal-Ziffer zusammengefasst.

vollziehbarere, Hochsprachen werden für die Ausführung auf dem Computer über einen *Compiler* übersetzt in Maschinencode, den dieser verarbeiten kann. Die Rückübersetzung von binärem Code in die Hochsprache ist hingegen nicht trivial und häufig nicht vollständig machbar. Aufgrund des Einwegfunktion-Charakters der Übersetzung von Hochsprache zu Maschinencode wird die logische Hochsprache üblicherweise „Quellcode“ oder englisch „Source Code“ genannt.

Quellcode beziehungsweise die Programmierung in „höheren Programmiersprachen“ ist also schon eine von den technischen Schaltungen abstrahierende Form, dem Computer Befehle zu geben. Sie erfolgt durch (meist) englische Kommandos und funktioniert formalsprachlich. Im Gegensatz dazu liegt Maschinencode recht nahe an der elektronischen Funktionsweise des Computers.

Der Exkurs in die Genese der Software zeigt, dass die virtuelle Repräsentation der Programmierung sich langsam von der materiellen Verdrahtung hin zum verbaltextlichen Aufruf von Befehlen entwickelt hat. Dabei müssen Programmierer*innen in höheren Abstraktionsebenen nicht mehr in Schaltungen denken, sondern schreiben logische Befehlsanordnungen, die vermittelt über die tiefer liegenden Ebenen in Schaltungsvorgänge übersetzt werden.⁶

UNIX, die „Mutter“ von GNU/Linux

Neben den genannten historisch-technischen Bedingungen, also der anfänglichen Verwandtschaft zwischen Programm-Aufruf und Programmierung, begünstigten ökonomische und soziale Gegebenheiten die Entstehung einer Praxis offener Software-Programmierung, die ich im Folgenden umreißen werde.

Aus verschiedenen Gründen spielte für die Entstehung der Freie-Software-Bewegung und die Entwicklung des Internets das Betriebssystem UNIX eine zentrale Rolle. Das Betriebssystem UNIX war von Anfang an ein sehr funktionsreiches Betriebssystem, das das gleichzeitige Arbeiten verschiedener Personen unterstützte und damit die Ressourcenverteilung in An-

⁶ Hingegen wird von Computer-Pionier Alan Turing sogar gesagt, er habe lieber den binären als den dezimalen Maschinen-Ausdruck gelesen (vgl. Hodges 1983: 399; zit. in Kittler 1993).

betracht der damals noch teuren Rechenzeit zu verwalten ermöglichte.⁷ UNIX fand daher schon recht früh eine relativ große Verbreitung und erfreute sich auch und besonders an Universitäten besonderer Beliebtheit. Der akademische Kontext der Experimentierfreudigkeit traf dabei auf den günstigen Umstand, dass aufgrund einer kartellrechtlichen Auflage der Hersteller AT&T das Betriebssystem UNIX nicht kommerziell vermarkten durfte. Daher war der Quellcode des Systems sehr günstig zu erwerben, was wiederum für die Universitäten von großem Interesse war – nicht zuletzt, um die Funktionsweise der Computersysteme zu studieren und zu lehren (vgl. ausführlich Grassmuck 2004: 214 ff.).

Die Neuartigkeit des Computers, die Unausgereiftheit des Produkts und die starke Verwendung an den Universitäten begünstigten die aktive Auseinandersetzung mit dem Quellcode, also die aktive Programmierung am Computer und folglich auch am Betriebssystem. Dies kam zusammen mit dem zunehmenden Aufbau von Netzwerkverbindungen zwischen den Computern der Universitäten und der dadurch entstehenden Möglichkeit, so über Probleme und Code-Fragmente zu diskutieren. Nachdem anfangs noch Programmcode über Disketten und per Post ausgetauscht wurde, ermöglichte eine computernahe Kommunikation neben dem Austausch von Argumenten auch den niederschweligen Austausch von Computercode als Gegenstand einer neuen Art von Diskussion über Softwarecode.

Die Computer-Nutzer*innen waren es also gewohnt, an der Software, dem Betriebssystem, herumzubasteln und Probleme und Lösungen wurden über das UseNet ausgetauscht und schließlich auch Code-Erweiterungen an den Hersteller von UNIX zurückgesendet. Nur so konnte man sichergehen, dass bei der nächsten Version nicht derselbe Fehler nochmal eigens ausgebügelt werden musste, sondern die eigenen Verbesserungen im Update enthalten waren.

Dadurch entstanden also eine Praxis der Auseinandersetzung mit dem Quellcode der Software und eine Art Gewohnheitsrecht der Nutzer, am Quellcode zu arbeiten und auch daran zu lernen. Wie oben erläutert handelt es sich dabei um eine formal-sprachliche Befehlskette eines Programms, die für das Verständnis und die Veränderung eines Programms insofern notwendig ist, als die für die Ausführung des Programms notwendige Übersetzung

⁷ Aufgrund der hohen Kosten der riesigen Rechenmaschinen gab es nur wenige Rechner, die sich viele User an den jeweiligen Institutionen teilten.

in Maschinencode das Erfassen der Funktionalität und die sinnvolle Modifikation äußerst schwierig macht.

1.2 Kommodifizierung von Software und der Reflex der Community

Anfang der 80er begann eine Wende in der Rolle von Software durch die Transformation von Software in ein eigenständiges Produkt. Bis dorthin wurde Software häufig als Bundle mit Support und Hardware verkauft und IBM war in dieser Hinsicht ein gigantischer Marktführer. Dies veranlasste das Kartellamt, IBM die Abspaltung der Software-Sparte aufzuerlegen, um kleineren Anbietern von Hardware und Software eine konkurrenzfähige Teilnahme am Markt zu ermöglichen. Daraufhin konnte sich ein Softwaremarkt entwickeln, es entstand zunehmend Wettbewerb unter konkurrierenden Herstellern und Software etablierte sich als eigenständiges Produkt (vgl. Grassmuck 2004: 203). Die Konkurrenz hatte zur Folge, dass die Funktionsweise der Software-Programme zum Betriebsgeheimnis der Hersteller wurde, der Quelltext als Rezept für konkrete Problemlösungsansätze zunehmend konsequent unter Verschluss gehalten und die Software rein in Maschinencode ausgeliefert wurde. Auch AT&T durfte nach einer Abspaltung der Softwareabteilung als „Unix Software Operation“ UNIX als Produkt vermarkten und fuhr folglich eine restriktivere Lizenzpolitik. Damit ging einher, dass der Zugriff auf den Quelltext für Nutzer*innen beschränkt wurde – Nutzer*innen, die jedoch aufgrund der oben genannten Umstände bisher zu einem guten Teil über die Bereitstellung von eigens erarbeiteten „Bugfixes“ und anderen Ergänzungen selbst aktiv mitentwickelt hatten. Diese fühlten sich folglich gewissermaßen enteignet (vgl. Holtgrewe/Werle 2001: 52).

Als Reaktion gründete Richard Stallman, Programmierer am Artificial Intelligence Lab des Massachusetts Institute of Technology, das Projekt GNU. Schon aus dem Namen kann man einen gewissen Trotz erkennen, das Akronym steht für „GNU’s Not Unix“. Das Projekt, das nicht UNIX ist, hat aber nichts weniger zum Ziel, als UNIX komplett nachzuprogrammieren. Um die erfahrene „Enteignung“ des durch die Community (mit)entwickelten Codes durch kommerzielle Unternehmen zu unterbinden, wurden Lizenzen entwickelt, die das Recht auf Weiterverbreitung und Veränderung sicherstel-

len anstatt zu begrenzen. Das Copyright, das dazu geschaffen wurde, lizenzierte Produkte davor zu schützen, anderweitig verwendet zu werden, wurde also dahingehend umgebaut, dass die Lizenz die weitere Nutzung explizit erlaubt. Dabei wurde das Copyright um die Funktion des *Copyleft* ergänzt, nämlich die Auflage, bei Wiederverwendung des Quellcodes die Folgeprodukte unter dieselbe (freie) Lizenz zu stellen, was als eine Art „Hack“ des Copyright betrachtet werden kann – eine Aneignung desselben für die eigenen Zwecke durch leichte Veränderungen (vgl. ebd.: 53f.).

Dieser Hack des Copyrights und die Schaffung einer Spielwiese für eine Computer-Avantgarde wurden dabei verknüpft mit der Verlautbarung einer gesellschaftlichen Wunschvorstellung: die allgemeine Zugänglichkeit von Informationen für mündige Bürger*innen, die damit etwas anfangen können und wollen. Stallman gründete die Free Software Foundation und trat folglich dafür ein, allen Nutzer*innen von Software vier „Freiheiten“ einzuräumen: (1) unbeschränkte Nutzung der Software, (2) die Möglichkeit, den Quelltext einzusehen, (3) zu modifizieren und (4) weiterzugeben (vgl. Stallman 2002).

1.3 Popularisierung und Expansion offener Software-Produktion

Die Freie-Software-Lizenzen bildeten eine rechtliche Grundlage, damit gemeinschaftlich entwickelter Code nicht apropriiert wird. Somit sollten alle, die ihren Code zu einer gemeinsam produzierten Software hinzufügen haben, durch die Anwendung der *freien* Lizenz sicher sein, dass sie dieses Programm auch zukünftig in der gewohnten Weise weiter nutzen und ergänzen dürfen. Im Gegensatz zu der sich etablierenden Praxis der kommerziellen Softwareproduktion wurde hierfür auch der Quelltext zur freien⁸ Verwendung veröffentlicht, da dieser die Basis zur Modifikation darstellt (s. Abschnitt 1.1). Zu den GNU-Programmen, die nach und nach die verschiedenen nachprogrammierten Befehle der UNIX-Welt umfassten, kam schließlich noch ein essenzielles Detail hinzu, der Kernprozess. Linus Torvalds schrieb

⁸ im Falle von Copyleft-Lizenzen mit der oben beschriebenen Freiheits-Einschränkung, modifizierte Varianten ebenfalls unter eine freie Lizenz stellen zu müssen

ein kleines Betriebssystem, das die Hardwarekomponenten steuern konnte, auch Kernel genannt (von Kernprozess). Die Veröffentlichung unter einer „freien“ Lizenz ermöglichte es nun, die UNIX-Programme von GNU mit diesem Kernprozess zu kombinieren, und somit war die Basis geschaffen für ein sehr umfängliches Betriebssystem, das komplett unter freien Lizenzen entwickelt war. Dieser Kernel namens „Linux“ wird heute weitläufig als Synonym für das gesamte Betriebssystem verwendet, während der Gründer von GNU bis heute auf die Bezeichnung GNU/Linux Wert legt (vgl. Grassmuck 2004: 223 ff.). Die computeraffine Avantgarde hatte sich somit in kollaborativer Arbeit ein eigenes Software-System geschaffen, das sie frei nutzen, gemeinschaftlich weiterentwickeln und nach Belieben an ihre Bedürfnisse anpassen konnte. Insbesondere war dies auf den sich vermehrt ausbreitenden *Personal Computern (PC)* lauffähig, die auch in einer Privatwohnung Platz fanden – im Gegensatz zu den bis dato üblichen Großrechnern. Somit war also auch die Hardware-Basis für die Software-Tüfteleien zunehmend für Privatpersonen erschwinglich. Computer wurden durch die Emergenz eines Marktes für den PC also zunehmend ein Produkt, das auch für Nutzer*innen interessant war, die keine professionellen Programmierer*innen waren. In dieser Phase der Computer Entwicklung traten also erst Nutzer*innen auf, die zunächst keine Programmier-Kenntnisse hatten. Bemerkenswerterweise hatte aber Microsoft durch einen genialen Coup mit IBM schnell eine überragende Marktführerschaft erlangt, und GNU/Linux fristete folglich im Bereich des Heim-PC lange Zeit ein Nischen-Dasein.⁹

*Wachsende Bedeutung von Laien-Nutzer*innen*

Jenseits von Linux wurde FLOSS auch für proprietäre Systeme (z.B. Microsoft Windows) entwickelt. Einen Eckpunkt der weiteren Ausbreitung von „Freier“ Software – oder „Open Source Software“, wie es im Sinne einer besseren Vermarktung zunehmend gebrandet wurde – stellte die Offenlegung des Quellcodes des Browsers „Netscape Navigator“ im März 1998 dar. Dies war bis 1996 der führende Internet-Browser, bevor Microsofts Internet Explorer sich zunehmend durchsetzte. In Anbetracht der sich abzeichnenden

⁹ Aus Platzgründen kann ich dieses spannende Detail leider nicht weiter ausführen, es gibt aber verschiedene Literatur über die Entstehung und den Erfolg von Microsoft. Die Geschichte über den entscheidenden Deal zwischen IBM und Microsoft, der den Grundstein für die monopolistische Marktführerschaft von Microsoft legte, findet sich beispielsweise bei Wallace und Erickson (1993) oder Grassmuck (2004: 204 ff.).

Niederlage im Konkurrenzkampf entschied sich Netscape dazu, den Quellcode des Netscape Navigators freizugeben (vgl. Spiegel 2006: 23). Mit dem Netscape Navigator und dem später darauf aufbauenden Browser Mozilla Firefox wurde ein populäres Stück Freie/Open Source Software von einem großen Kreis von Nutzer*innen verwendet. Durch FLOSS-Programme wie Mozilla Firefox, die auch unter Microsoft Windows laufen, öffneten sich also neue, weniger technikaffine Nutzerkreise, die sich weniger mit der Idee Freier Software oder gar der Funktionsweise ihres Computers auseinandersetzen (wollen). Nachdem die Freie-Software-Bewegung also aus einem Experten-Kontext von meist akademischen Programmierer*innen entstanden ist, fand in der Freigabe des Quellcodes des weit verbreiteten Netscape Navigator ein FLOSS-Produkt den Weg auf die Microsoft-PCs ganz normaler Laien-User.

GNU/Linux spielt im professionellen Server-Bereich von jeher eine tragende Rolle, im Anwenderbereich fristete das System aber lange Zeit eine marginale Rolle. Im Jahr 2004 wurde ein Projekt gegründet, das zum Ziel hatte, GNU/ Linux stärker in den Anwenderbereich zu bringen, um Microsoft Windows vom Thron der absoluten Marktführerschaft zu heben.¹⁰ Dazu startete „Ubuntu“ mit dem Motto „Linux For Human Beings“ und adressierte somit explizit Nicht-Expert*innen, also Menschen, die Laien-User sind. Hatte sich mit Internet-Browsern und anderen Anwendungsprogrammen die Freie Software Community also schon einen Kreis von Laien-Usern erschlossen, strebte Ubuntu an, auch die darunter liegende Schicht des Betriebssystems für Laien nutzbar zu machen – und, wie ich im Laufe der Arbeit erörtern werde, damit Laien auch strukturell als mögliche Beitragende im Sinne einer offenen Softwareproduktion zu verstehen (vgl. Kap. 8).

Wenige Jahre vorher (2001) manifestierte sich in der Gründung von Arch Linux ein Gegentrend zum Paradigma der Usability: Statt den Usern die Funktionsweise der Computer durch einfache Bedienoberflächen und Assistenten zu verbergen – und dadurch die Komplexität des Systems zusätzlich zu erhöhen –, soll die Technik vielmehr „einfach“ sein und User sollen die Möglichkeit haben, sie besser zu verstehen. Beispielhaft stelle man sich dazu die grafische Oberfläche im Gegensatz zur Kommandozeile vor: Die grafische Oberfläche generiert eine neue Abstraktionsebene und beinhaltet komplexe Vorgänge, die Anwendungen und Prozesse visualisieren und Mausbe-

10 <https://bugs.launchpad.net/ubuntu/+bug/1>

wegungen in Programm-Befehle übersetzen müssen. Dabei verschwindet das einzelne Programm mit seinen Parametern in den Hintergrund.¹¹

Ubuntu und Arch stehen also für ein unterschiedliches Verständnis von Einfachheit (einfache Nutzung vs. einfache Wartung) und bilden interessante Vergleichsfälle zur Betrachtung der Rolle von Expertise und Laien innerhalb der Community. Die Arch-Philosophie wendet sich explizit gegen vorkonfigurierte Linux-Systeme und propagiert stattdessen ein ganz basales System, das der Nutzer frei konfigurieren und an seine Bedürfnisse anpassen kann. Das bringt mit sich, dass User sich eindringlicher mit den zugrundeliegenden Konzepten beschäftigen müssen oder eben – dürfen. Dies hat aber auch den Effekt, dass diese (zwangsläufig) einiges über die Funktionsweise des Systems lernen. Bemerkenswerterweise erfreut sich auch Arch wachsender Beliebtheit. Jedoch kommen dort infolge der Popularisierung der Community Stimmen auf, die in der Konsequenz ein sinkendes Qualitäts-Niveau beklagen und nicht begeistert davon sind, eine steigende Präsenz von technisch weniger versierten Nutzer*innen beraten zu müssen.

Hier zeigen sich also unterschiedlich ausgerichtete normative Ordnungen der Communities, die sich punktuell auch in Konflikten zwischen Mitgliedern der Communities manifestieren. Auf dieser individuellen Ebene zeigen sich unterschiedliche Wertungen und Ansprüche, die ich im Laufe dieser Arbeit als Teil regulierender epistemischer Strukturen der verschiedenen Gemeinschaften untersuchen werde. Damit einher gehen folglich verschiedene Grade an Bewertung und Akzeptanz unterschiedlicher Beiträge sowie divergierende Kompetenz- oder Expertisestufen von beitragenden Usern.

Eine historische Bedingung der Herausbildung unterschiedlicher epistemischer Regime beruht auf der Durchmischung einer weitgehenden Experten-Community durch Laien, die durch die zunehmende Popularisierung des Projekts „Open Source Software“ stattfindet. Ubuntu und Arch stehen für verschiedene Ansätze, auf die Veränderung zu reagieren: eine Ausrichtung und Adressierung der Laien auf der einen und eine explizite Abgrenzung der Expert*innen von den Laien auf der anderen Seite. Bemerkenswert ist dabei, dass diese Differenzierung verschiedener Regime hier auf der strukturellen Differenz zwischen einer historisch gewachsenen Experten-Community und einem ideologischen Anspruch beruht, der eine Aufklärung der unmündigen Nutzer*innen anstrebt und impliziert, dass das Lesen und Verändern der

¹¹ Exemplarisch für die Weiterentwicklung grafischer Oberflächen sei hier auf die Entwicklung der Computer-Maus verwiesen (vgl. English/Engelbart/Berman 1967).

Software im Interesse aller Computer-User sei oder sein sollte. Ubuntu und Arch stehen dabei auch für verschiedene Ansichten bezüglich des Umgangs mit Komplexität – verbergen (Usability) vs. möglichst verständlich gestalten („Keep it simple, stupid“) –, für unterschiedliches Sendungsbewusstsein und nicht zuletzt für die Motivationen, die Verbreitung eines „freien Systems“ als Empowerment voranzutreiben (Ubuntu) oder aber sich einen kleinen Freiraum für sich zu bewahren (Arch). Daher bilden diese beiden Pole einen interessanten Untersuchungsgegenstand für das Verhältnis von Expert*innen und Laien in der offenen Software-Produktion. Bevor ich näher auf die theoretischen Implikationen und die empirische Betrachtung eingehe, werde ich verschiedene normative Implikationen von Technik und insbesondere Software diskutieren, die aus einer soziologischen Perspektive die ideologische Motivation ergründen, Software transparent und modifizierbar zu gestalten.

1.4 Legitimationsschwierigkeiten der Expert*innen

Die Freie-Software-Bewegung entstand also zunächst aus der Gewohnheit heraus, Software eigenmächtig zu verändern. Sie begründete ihr Anliegen der Zugänglichkeit des Software Source Codes von Anfang an auch mit moralischen Argumenten der Transparenz und der Kontrollmöglichkeiten der User und kritisierte damit die Legitimität der Enthebung der Software-Programmierung aus dem Kontext ihrer User. Dies spiegelt sich auch wider in den langwierigen Debatten über den Begriff der „Software-Freiheit“ (vgl. dazu Berry 2008: 98 ff.). Bis heute ist den einen der Begriff *Freie Software* zu ideologisch, für die anderen vernachlässigt der Begriff *Open Source Software* zu sehr die politische Grundierung der Bewegung.¹²

„Frei“ bedeutet dabei, wie die Free Software Foundation betont, „free as in freedom“. Damit werden neben der Möglichkeit der Weitergabe des Quelltextes (Informationsfreiheit) explizit die Kontrollmöglichkeiten der Software-User adressiert: “[...] the freedom to change a program, so that you can con-

¹² Siehe insbes. <https://www.gnu.org/philosophy/open-source-misses-the-point.en.html> (letzter Aufruf 8.3.2017).

trol it instead of it controlling you; for this, the source code must be made available to you” (Stallman 1986).

In der Zurückweisung einer exklusiven Hoheit einzelner Firmen über die Funktionsweise der Software und der damit verbundenen Schließung des Wissens lässt sich eine Analogie ziehen zur Kritik der Autorität der Wissenschaften in der zweiten Hälfte des 20. Jahrhunderts, die Collins und Evans (2002) in ihrem Aufsatz „Third Wave of Science Studies“ beschreiben. Insbesondere infolge Thomas Kuhns (1970) Dekonstruktion „wissenschaftlicher Revolutionen“ schwindet ihrer Argumentation zufolge die Selbstverständlichkeit des wissenschaftlichen Positivismus. Die darauf folgende sozialkonstruktivistische Wissenschaftskritik der Sociology of Scientific Knowledge und der Science and Technology Studies legt ihr Augenmerk dezidiert auf die Entstehung wissenschaftlichen Wissens und dekonstruiert so dessen Objektivität. Dadurch stellt sie die Gewissheit wissenschaftlicher Tatsachen infrage – und in der Konsequenz auch die Legitimation wissenschaftlich begründeter Entscheidungen, beziehungsweise die Autorität wissenschaftlicher Expert*innen in Bezug auf politische Entscheidungsprozesse.

Diese Dekonstruktion des Vertrauens in wissenschaftliches Wissen bezeichnen die Autoren als die ersten beiden „Wellen“ der Science Studies. Sie führte zu einer *Legitimationskrise* („Problem of Legitimacy“) der wissenschaftlich ausgewiesenen Expert*innen. Infolge dieser Problematisierung kam es zu einer zunehmenden Erweiterung des Kreises der an der Beratung und Diskussion politischer Entscheidungsprozesse Beteiligten. Die Beteiligung von nicht-zertifizierten Expert*innen in politische Entscheidungsprozesse, also beispielsweise Menschen mit Erfahrungs- und Feldwissen, hat in der Form von partizipativen Verfahren wie Bürgerbeteiligung über Open Data und Open Government bis hin zu Multi-Stakeholder-Initiativen eine gewisse Etablierung erfahren.

Jedoch ergibt sich aus der Erweiterung des Kreises der Beteiligten die Frage, bis zu welchem Maße die Erweiterung sinnvoll ist, also wer in welchen Prozessen und Verfahren beteiligt sein sollte. Vermieden werden sollte, dass sich Gremien in Beratungen verlieren oder letztlich wenig hilfreiche oder gar sachlich falsche Argumente von Teilnehmer*innen mit unzureichendem Wissen in die notwendigen politischen Entscheidungsprozesse einbezogen werden. Dies nennen Collins und Evans das *Erweiterungsproblem* („Problem of Extension“), sie argumentieren für eine Rückbesinnung auf die Fähigkeiten wissenschaftlicher Expert*innen und eine reflexive Berücksichtigung des Wissens derer, die spezifisches Wissen aufgrund ihrer Exper-

tise oder besonderes Erfahrungswissen besitzen. Sie sehen die Gewissheit wissenschaftlicher Tatsachen vor allem in der Phase der Konstruktion des Wissens infrage gestellt, also in jungen wissenschaftlichen Debatten, wo noch mehrere konkurrierende Perspektiven und Erkenntnisse existieren. Im Laufe der Zeit etabliert sich demnach aber ein wissenschaftlicher Konsens, der gesichertes Wissen darstellt.

Legitimationskritik der Free Software Foundation

In Analogie zur Krise der Wissenschaft und der Expert*innen zeigt sich im Bereich der Software eine Kritik der Legitimation der exklusiven Herstellung der Software durch die Technik-Hersteller*innen. Diese wird insbesondere formuliert und vertreten durch die *Free Software Foundation*. Sie tritt ein für einen Zugang zum in Software vergegenständlichten Wissen und fordert den uneingeschränkten Zugriff auf das zugrundeliegende Wissen, den Quelltext. Diese Kritik steht zunächst im Zusammenhang mit der Forderung nach Informationsfreiheit, schließt aber im weiteren Sinne auch die Frage der Akzeptanz technologischer Entwicklung ein im Sinne einer Skepsis gegenüber den Einflussmöglichkeiten der Technik und damit verbunden der sie gestaltenden Akteure.

In ihrer Monografie *Rethinking Expertise* begründen Collins und Evans (2007) das Legitimationsproblem normativ mit der Notwendigkeit gesellschaftlicher Akzeptanz technischer Entwicklungen:

The public have the political right to contribute, and without their contribution technological developments will be distrusted and perhaps resisted. This is what we called the “Problem of Legitimacy.” (ebd.: 113)

Diese Kritik steht also sowohl im Zusammenhang mit der politischen Gestaltung der Gesellschaft durch technische Entwicklungen als auch mit der politischen Forderung nach Teilhabe und Transparenz von Wissen, wie im vorigen Abschnitt ausgeführt.

Die Legitimitätskritik der Free Software Foundation entzieht sogenannter proprietärer Software – also Software ohne die Möglichkeit der Beteiligung einer Öffentlichkeit (eines jeden Users) – pauschal jegliches Vertrauen und somit ihre Legitimität. Schon 1986 schreibt Richard Stallman in der ersten Ausgabe des GNU Bulletin:

The word “free” [...] does not refer to price; it refers to freedom. First, the freedom to copy a program and redistribute it to your neighbors, so that they can use it as well as you. Second, the freedom to change a program, so that you can

control it instead of it controlling you; for this, the source code must be made available to you. (Stallman 1986)

Stallman fokussiert bei seiner Formulierung also zunächst auf die Freiheit der Zirkulation von Informationen, was zuerst eine Frage des Zugriffs oder der Rezeption von Wissen darstellt. Das zweite Argument fokussiert die wechselseitige Kontrolle von User und Programm und macht daraus einen starken Punkt: Wenn User das Programm nicht verändern und damit kontrollieren können, werden diese kontrolliert und gesteuert.

Anders als Wissenschaft, die, *vermittelt* über die Unterstützung politischer Entscheidungsprozesse durch wissenschaftliches Wissen, eine gewisse gesellschaftliche Macht ausübt, wirkt sich die Gestaltung von Software direkt auf die Nutzung aus. In beiden Fällen steht das Vertrauen in die „Hüter*innen“ des Wissens – die zertifizierten Expert*innen der Wissenschaft und der Softwareherstellung – zur Disposition. Dabei geht es nicht zuletzt um die Transparenz der Genese von Wissen und der Legitimation des Expertenstatus.

Ausgehend von der Annahme, dass Software in zunehmenden Bereichen des Alltags zentrale Funktionen übernimmt, steckt in der Kritik der Legitimation entbetteter Softwareherstellung eine politische Frage der Gestaltung von Gesellschaft. Das Vertrauen in die legitime Produktion von Wissen hängt hier also zusammen mit der Legitimation einer über Wissen vermittelten Einflussnahme auf die Gesellschaft. Diese Einflussnahme als normative Implikationen von Technik ist Gegenstand der wissenschaftlichen Diskussionen, die ich im nächsten Abschnitt wiedergebe.

2 Implikationen von Technik und Software

Nachdem ich im Zusammenhang mit der Entstehung der Freie-Software-Bewegung die inhärente Kritik der alleinigen Gestaltungsmacht durch privilegierte Software-Hersteller*innen beschrieben habe, werde ich in diesem Kapitel zeigen, wie in verschiedenen wissenschaftlichen Disziplinen ebenfalls die Gestaltungsmacht der Hersteller*innen und das Kontrollpotenzial von Software problematisiert wird.

Die Problematisierung der Gestaltbarkeit von Software steht also im Zusammenhang mit der Frage ihrer Beschaffenheit und Wirkung. Die Idee der Rolle von Normen in der Gestaltung von Technik wurde verschiedentlich diskutiert; prominent stellt Latour (1991) fest, dass in Technik Handlungsprogramme eingeschrieben werden können, wodurch Normen in ihrer Wirksamkeit durch Technik verstärkt werden. Während schon einfache Techniken wie Tempohemmschwellen („Speed Bumps“) (vgl. Latour 1992) als „Härter“ gesellschaftlicher Normen erscheinen (vgl. Degele 2002), fokussieren jüngere Diskurse insbesondere auf Software-basierte Technik. Im Gegensatz zu schweren Schlüsselanhängern, die Hotelgäste dazu nötigen, den Schlüssel an der Rezeption zu lassen (vgl. Latour 1991), Brücken, die durch ihre Bauhöhe die Durchfahrt von Bussen verhindern und damit potenziell bestimmte Bevölkerungsgruppen ausschließen (vgl. Winner 1980), oder Rasierapparate, die durch ihr Design Gender-Stereotype reproduzieren (vgl. Van Oost 2003), sind die normativen Einschreibungen in Software-basierte Technik auf fundamental andere Weise programmierbar, (un-)sichtbar und schließlich wirksam.

Die aufgezählten Artefakte haben eine statische Beschaffenheit, die im Wesentlichen auch gut sichtbar ist, wenngleich die normativen Prämissen nicht unbedingt offensichtlich sind (und auch den Konstrukteur*innen nicht notwendigerweise bewusst sind). Im Gegensatz dazu ist die Funktionsweise eines Software-Programmes dynamisch veränderbar und bleibt unter der Oberfläche der Technik verborgen. Insbesondere stellt die grafische Oberfläche eines Software-Programms eine reine Projektion des Programms dar und ist in gewissem Maße unabhängig von den Prozessen, die unter der Oberfläche im Innern des Computers passieren. Dieses Phänomen verstärkt sich bei

Alltagsgegenständen, die mittels Software und Internet-Verbindung mit der kaum fassbaren potenziellen Menge von Computern im *World Wide Web* verbunden sein können.

Die regulierende Wirkung von Technik wird vor dem Hintergrund der wachsenden Rolle Software-basierter Technik in jüngeren wissenschaftlichen Diskursen vermehrt diskutiert. Diese fasse ich im den folgenden Abschnitten zusammen. Die Problematisierung der Kontrollwirkungen von Technik gesteht Technik eine gewisse determinierende Wirkung zu. Dies hängt zusammen mit der besonderen Eigenschaft von Software-basierter Technik gegenüber anderer Technik, die ich im anschließenden Kapitel soziologisch charakterisieren möchte.

2.1 Techno-Regulation und die Rolle von Software

Ein Forschungszweig, der sich mit der beeinflussenden Wirkung von in Technik und den darin eingeschriebene Normen beschäftigt, ist der Bereich der sogenannten *Techno-Regulation*. Zunächst versammeln sich unter diesem Begriff vornehmlich Rechtswissenschaftler*innen und Rechtsphilosoph*innen, die sich mit der regulierenden Wirkung von Technik und den darin eingeschriebenen Normen beschäftigen und diese der Funktion und Wirkung von Gesetzen gegenüberstellen (vgl. Brownsword 2005). Dabei wird festgestellt, dass Softwarecode eine Art Gesetz des Cyberspace darstellt (vgl. Lessig 1999), da in Softwarecode die Regeln des sozialen Miteinanders innerhalb der digitalen Infrastrukturen nicht nur festgelegt, sondern aufgrund der unmittelbaren Gestaltung der virtuellen Welt durch die eigens in Software beschriebenen Regeln zugleich auch *wirksam durchgesetzt* werden. Beispielsweise definiert Softwarecode, wie viele Menschen sich gleichzeitig in einem bestimmten Chat unterhalten können, ob sie das anonym machen können oder nicht, oder auf welche Art man Inhalte anderer teilen oder auch bewerten kann. Die in Software definierten Möglichkeiten schreiben so die Bedingungen der Nutzung fest, ohne dass sichtbar wird, welche anderen Gestaltungsmöglichkeiten denkbar wären und auf welchen Prämissen die Beschränkungen beruhen. Die Beschaffenheit der Software strukturiert die Interaktion der User auf sehr grundlegende Weise.

Dabei steht in den Diskursen um Techno-Regulation häufig die Frage der Rechtmäßigkeit regulierend wirkender Technologien im Zentrum der Diskussion. Zwar haben Technik-Hersteller*innen – anders als legislative Organe eines Rechtsstaates – keinerlei demokratische Legitimation, wirksame Regeln zu definieren, dennoch stellen verschiedene Autor*innen wiederholt fest, dass sich in Technik eingeschriebene Verhaltensregeln leichter forcieren lassen. Während bei gesetzlichen Vorschriften deren Verstoß in der Regel erst nachträglich geahndet wird, kann eine technische Umsetzung von Regeln alternative Handlungsmöglichkeiten ausschließen (vgl. Hildebrandt 2008b; Koops 2007; Leenes/Koops 2005). Obgleich diese rechtsphilosophischen Erwägungen an dieser Stelle ausgeblendet bleiben, gibt es einige wichtige Aspekte dieser Diskussion, die eine besondere Rolle von Software als spezifische Form von Technik nahelegen.

Konstitutive und regulative Regeln

Hildebrandt (2008b) diskutiert die Unterscheidung von regulativen und konstitutiven Regeln, die sowohl im Recht als auch in der Technik unterschieden werden können. Demnach sind Regeln konstitutiv, wenn die Befolgung eine notwendige Bedingung für eine Handlung ist; beispielsweise wird eine Eheschließung nur dann vollzogen, wenn die dafür notwendigen (und damit konstitutiven) Regeln eingehalten werden. Regulativ sind Regeln, wenn sie eine bestehende Handlung zu steuern versuchen – die aber unabhängig von der Befolgung der gewünschten Regeln durchgeführt werden kann. Dies präzisiert die oben genannte These, dass in Technik eingeschriebene Regeln häufig leichter durchgesetzt werden können, da Recht grundsätzlich eher reaktiv ist und *ex post* durchgesetzt werden muss, in Technik festgelegte Regeln für das technische Handeln aber konstitutiv sein können (vgl. Hildebrandt 2008a,b). Diese zunächst generelle Annahme wird in der Literatur an verschiedenen Beispielen problematisiert. Dabei kommen vermehrt Beispiele mit Software-basierten Technologien zur Anwendung, die offenbar noch weitere Implikationen bergen.

Die *konstitutiven* Regeln in Technik besitzen einen hohen Grad an Einflussnahme und können bestimmte Bedingungen erzwingen (beispielsweise eine maximale Durchfahrtshöhe wie oben bei Winner (1980)); regulative Regeln sind dabei eher durch Informationen oder allenfalls durch Anreize implementiert (vergleichbar zu Hinweisschildern und Schlüsselanhänger bei Latour (1991), s.o.). Dabei gibt es bei der Gestaltung von Technik Spielräume, wie stark die Vorgaben sind und wie viele Handlungsspielräume den

Nutzer*innen bleiben. Technik kann somit mehr oder auch weniger Kontrolle (im Sinne von Steuerung) ausüben. Ein prominentes Beispiel thematisiert die Umsetzung einer Anschnallpflicht. Ein Smart-Car kann entweder über eine Leuchte dem/der Fahrer*in eine optische Rückmeldung über den fehlenden Anschnallgurt geben oder im Falle des fehlenden Gurts schlicht den Motor nicht starten (vgl. Hildebrandt 2009). Eine weitere Möglichkeit ist, dass das Auto über laute Geräusche einen starken Anreiz gibt, sich anzuschnallen. Im Falle des Nicht-Startens des Autos ist die Anschnallregel also konstitutiv für das Starten des Autos, bei der Warnleuchte und dem Warnton nicht. Die Warnleuchte und der Warnton unterscheiden sich jedoch stark in Bezug auf die Eindringlichkeit des Hinweises. Während die Leuchte eine Information darstellt, verkörpert das Warnsymbol schon eher eine negative Sanktion und gibt einen starken Anreiz, die Anschnallregel zu erfüllen, um damit das nervende Geräusch abzustellen.¹³

Eine spezifische Rolle von Software kristallisiert sich aber erst heraus durch die Kombination zweier Formen von Kontrolle, die in einem besonderen Maße durch Software-basierte Technologien möglich werden. Im Folgenden werde ich daher zunächst ein Beispiel der Literatur aufgreifen, das Kontrolle im Sinne von Sichtbarkeit, Speicherbarkeit und damit Überwachbarkeit thematisiert, und anschließend ein weiteres Beispiel, das die Protokollierbarkeit von Verhalten mit Kontrolle im Sinne von Einflussnahme kombiniert.

Kontrolle im Sinne von Sichtbarkeit

Ein weiterer Strang der Diskussion dreht sich um die technischen Kontrollmöglichkeiten im Sinne einer Überwachung der Aktivitäten, aber auch im Sinne von Beeinflussung: Leenes und Koops (2005) vergleichen das traditionelle Funkwellen-Radio (UKW) mit IP-basiertem Webradio und stellen fest, dass trotz derselben Funktionalität auf Seiten der User – das Hören einer Radio-Sendung – ein signifikanter Unterschied besteht in Bezug auf die Möglichkeiten der Nachvollziehbarkeit der Hörgewohnheiten der User durch den Radiosender oder die Betreiber der entsprechenden Infrastruktur (Kontrolle im Sinne von Überwachung). Funk bietet zunächst keine Möglichkeit zu überprüfen, wer alles die ausgestrahlten Funkwellen in Audio-Signale übersetzt, ein Webradio-Anbieter kann aber die Zugriffe der verschiedenen

13 Zu verschiedenen Einflussformen von Technik vgl. ausführlich Berlin Script Collective (2018).

Internetadressen der Rezipienten detailliert protokollieren und verschiedene Akteure wie die Internet-Service-Provider können dies ebenfalls. Aus der reinen User-Perspektive ist das zunächst nicht unbedingt ersichtlich. Jedoch können IP-Adressen häufig auf die individuelle Nutzung von Bürger*innen zurückgeführt werden. Die politischen Implikationen zeigen jüngste Forderungen der amerikanischen Regierung, die IP-Adressen der Besucher*innen einer Seite zu erlangen, die zu Demonstrationen gegen Donald Trump aufgerufen hatte.¹⁴

Kontrolle im Sinne von Steuerung

Die individuelle Bedeutung unsichtbarer Kontrollmöglichkeiten durch Technik und deren Einschreibungen zeigt außerdem die Auseinandersetzung mit datenbasierten Profiling-Technologien. Hildebrandt (2008c: 63) beschreibt dabei ein eindrückliches Beispiel unsichtbar wirksamer Skripte im Bereich des Online-Shoppings:

Imagine that my online behaviour is profiled and matched with a group profile that predicts that the chance that I am a smoker who is on the verge of quitting is 67%. A second profile predicts that if I am offered free cigarettes together with my online groceries and receive news items about the reduction of dementia in the case of smoking I have an 80% chance of not quitting. This knowledge may have been generated by tobacco companies, which may use it to influence my behaviour.

Hier wird die Möglichkeit beschrieben, im verborgenen Teil einer technikvermittelten Handlung einen Anreiz in Form eines Werbegeschenks mit einer Information zu ergänzen und somit eine hohe Wahrscheinlichkeit der Handlungsbeeinflussung zu erzielen – nämlich hier ganz konkret, *nicht* mit dem Rauchen aufzuhören.¹⁵ Eine elementare Rolle für die Wirksamkeit der Beeinflussung spielt hier die Erfassung umfangreicher Daten über die Gewohnheiten des Users, die erst eine derartige Berechnung möglich machen.

14 <https://www.nytimes.com/2017/08/18/opinion/justice-department-dreamhost-site-trump.html>, letzter Aufruf 21.8.2017

15 Die empirische Relevanz derartiger Praktiken zeigt unter anderem eine Reihe von Studien zum Einfluss von Profiling auf die Grundrechte (vgl. Creemers/Guagnin/Koops (2015). Insbesondere zeigen die Diskussionen um „Cambridge Analytica“ und deren Einflussnahme auf den US-Wahlkampf 2016 ein eindrucksvolles Beispiel für die politischen Einflussmöglichkeiten dieser Art, siehe beispielsweise <https://www.theguardian.com/politics/2017/mar/04/nigel-oakes-cambridge-analytica-what-role-brexit-trump> (letzter Aufruf 22.8.2017).

Der wesentliche Kritikpunkt zielt hierbei auf die Informationsasymmetrie zwischen Endanwender*in und Betreiber*in des Technologie-Arrangements. Da dem User die Zielgenauigkeit der Manipulation der bereitgestellten Informationen kaum bewusst sein kann, wird der User in seiner Autonomie eingeschränkt. Die Intransparenz algorithmischer Prozesse verschleiert die informationellen Grundlagen der Entscheidungsfindung und vermindert somit die Autonomie des entscheidenden Subjekts (vgl. ebd.: 63). Das trifft im Grunde nicht nur auf informierte Entscheidungen zu, sondern generell auf durch Design von Technik beeinflusste Entscheidungen von Usern, da denkbare Alternativkonstruktionen und die eingebauten moralischen Implikationen nicht unbedingt sichtbar sind. Mit anderen Worten, nicht nur die Selektion und Konstruktion von Daten, sondern auch die materielle Gestaltung von Technik bedeuten eine Vorauswahl von Entscheidungsmöglichkeiten, die den Usern angeboten werden, und beeinträchtigen somit deren Autonomie.

Je nach Gestaltung der Technik werden also verschiedene Grade der Härte einer Regeldurchsetzung erwirkt und damit unterschiedlich stark Einfluss auf die Handlung der User genommen. Ein besonderes Augenmerk verdient dabei die Sichtbarkeit der eingeschriebenen Normen, da die wahrgenommenen Informationen eines Nutzers oder einer Nutzerin die Grundlage ihrer Entscheidungen bilden. Maßgeblich für die Kontrolle im Sinne von Überwachung und Steuerung sind aggregierte Daten, die zu Informationen verarbeitet werden und so die Wirkung der Technik beeinflussen. Dadurch wirken sie wieder auf ihre User zurück. Hierbei spielt Software eine zentrale Rolle, da durch Software vermittelte Prozesse leicht protokollierbar sind und Computerprogramme erst die Verarbeitung großer Datenmengen ermöglichen.

Die regulierenden Potenziale von Technologie wurden in der Techno-Regulation-Debatte schon diskutiert, sind aber in den letzten Jahren durch ihre wachsende Realisierung in Suchmaschinen und Sozialen Netzwerken in ihrer Tragweite sichtbar geworden. Folglich entstand in den letzten Jahren das Forschungsfeld der *Critical Algorithm Studies*, das ich im folgenden Abschnitt umreiße.

2.2 Software wird kritisch: Critical Algorithm Studies

In den 2010er-Jahren begann eine neue Welle sozialwissenschaftlicher Beschäftigung mit dem Thema Algorithmen und Fragen nach der „Materiality of Algorithms“ (Anderson 2012), „Politics of Algorithms“ oder „Governing Algorithms“¹⁶. In einem Essay zum Diskussionsanstieg für die gleichnamige Konferenz schreiben Barocas, Hood und Ziewitz (2013: 3): “To judge from the steady increase in algorithm-themed workshops, conference sessions, and media mentions over the past twelve months, algorithms have become a hot topic.”

Vor dem Hintergrund der wachsenden Software-basierten, sozio-technischen Infrastrukturen wie Internet-basierten Plattformen und sozialen Online-Medien tritt das wirksame Gestaltungspotenzial von Technik auf neue Weise zutage. Infolge der ersten Welle der oben referenzierten Case-Studies zu normativen Inskriptionen (vgl. Akrich 1992) in Brücken und Schlüsselanhängern bildet sich nun eine zweite Welle von Fallstudien zu Aspekten und Effekten Software-basierter Technologien, die zunehmend die Rolle der Algorithmen thematisieren.

In Anbetracht der beeindruckenden Wirkungen und Potenziale Software-basierter Technik kommen durchaus Zweifel auf an der strikten Zurückweisung des Technikdeterminismus. Wenngleich die von Technik-Enthusiasten erhoffte gänzliche Lösung sozialer Probleme durch Technik zurückgewiesen werden muss (Morozov 2013), so bleiben auch bei der Auflösung des Determinismus durch die Konstruktion von Handlungsspielräumen gewisse technologische Determinanten:

Affordance theory asserts a number of things, but I'd focus on its claims that technologies produce fields of action (including unexpected actions), but that not all actions are possible. The “not all actions” indicates something irreducible in each technological artefact that lets us say that in specific situations technologies do determine (as long as “determine” is understood as fields of action and not one single action). (Tim Jordan in: Neff/McVeigh-Schultz/Jordan 2012)

Ziewitz (2016) fasst auf lakonische Weise die „Structure of concerns“ einiger Studien zur Rolle von Algorithmen zusammen als ein Drama in zwei Akten. Im ersten Akt werden Algorithmen als mächtige und folgenreiche Akteure in

16 <http://governingalgorithms.org/>, letzter Aufruf 1.9.2018

vielfältigen Anwendungsfeldern konstatiert: “search engines, online news, education, markets, political campaigns, urban planning, welfare, and public safety”. Im zweiten Akt werden die Schwierigkeiten für Erklärungsansätze formuliert, wie diese Macht und der Einfluss durch die Algorithmen zustande kommen und konkret wirksam werden. Dabei wird die Undurchsichtigkeit und Unsichtbarkeit ihrer Wirkungsweise als weiteres Indiz für ihre Macht und Einflussnahme gedeutet.

Dabei ist gar nicht immer klar, ob unter Algorithmus alle dasselbe verstehen. Techniker*innen, Sozialwissenschaftler*innen und die breitere Öffentlichkeit haben da durchaus unterschiedliche Perspektiven.

In the case of algorithm, the technical specialists, the social scientists, and the broader public are using the word in different ways. For software engineers, algorithms are often quite simple things; for the broader public they are seen as something unattainably complex. For social scientists, *algorithm* lures us away from the technical meaning, offering an inscrutable artifact that nevertheless has some elusive and explanatory power. (Gillespie 2016: 18)

Nichtsdestotrotz besteht weitgehende Einigkeit, dass Algorithmen ein wichtiges Untersuchungsfeld darstellen, das aus den verschiedensten Disziplinen untersucht werden kann:

Some suggest that there are different ways of studying algorithms. For example, at a recent workshop a scholar suggested that there could be a technical approach that studies algorithms as computer science; a sociological approach that studies algorithms as the product of interactions among programmers and designers; a legal approach that studies algorithms as a figure and agent in law; and a philosophical approach that studies the ethics of algorithms. (Barocas/Hood/Ziewitz 2013: 14)

Kitchin (2017: 20) plädiert für eine “code/software studies’ perspective that studies the politics and power embedded in algorithms”. Er fasst die Schwierigkeiten der Forschung über Algorithmen zusammen in (1) schwierigen Zugang zu Algorithmen und ihre Undurchschaubarkeit (*Black Box*), (2) ihren heterogenen und eingebetteten Charakter, durch die unterschiedlichen Befehls-Ebenen (vgl. auch oben, S. 31) und verschiedenen Datenquellen, auf die sich Algorithmen beziehen, und schließlich (3) ihre Kontingenz aufgrund nicht-linearer Abläufe und häufig schwer vorhersehbarer Dateneingaben und ständiger Veränderung. Als Lösungsansatz schlägt er Forschungsmethoden vor wie den Quellcode und die dazugehörigen Kommentare zu studieren, selbst reflexiv Code zu produzieren oder Reverse Engineering zu betreiben, um aus binären Programmen die Funktionsweise zu rekonstruieren. Schließlich mündet die Aufzählung in eher klassische Methoden wie die ethnografi-

sche Beobachtung von und Interviews mit Programmierer*innen, die sozio-technischen *Assemblagen* zu eruieren sowie die Wirkweise „in der Welt“ zu betrachten (vgl. ebd.: 22 f.) .

Anderson (2012) hingegen fordert eine stärkere Berücksichtigung der Daten-Eingaben, da diese die Arbeitsgrundlage der Algorithmen darstellen: “To understand algorithms we need analyze their affordances, meanings, ethics, legal infrastructures, and computational processes, but we also need to understand the very material fuel that keeps them powered in the first place.”

Zu beobachten ist also eine verstärkte Fokussierung auf die handlungsbeeinflussenden Wirkungen von Technik und die zentrale Bedeutung von Algorithmen, nicht zuletzt auch durch deren verbreitete Integration in Technik. Dies steht für eine wachsende gesellschaftliche Problemwahrnehmung der Bedeutung von Software, einer offensichtlich schwer begreifbaren, aber wirkmächtigen Technik. Die Argumente der Legitimationskritik der Free Software Foundation, nämlich die Rolle von Software für Autonomie und Kontrolle – “the freedom to change a program, so that you can control it instead of it controlling you” –, spiegelt sich wider in den theoretischen und empirischen Problembeschreibungen verschiedener wissenschaftlicher Disziplinen, die sich aufgrund der zunehmenden Integration von Software in die alltägliche Lebenswelt empirisch mit deren mannigfachen Wirkungsweisen konfrontiert sehen.

Zentrale Fokuspunkte der Problematisierung bilden hierbei die Undurchsichtigkeit der Algorithmen, der verwendeten Daten sowie der Art der Verwendung der Daten. Diese Fokuspunkte sind auch Gegenstand der *Surveillance Studies*, die ich im nächsten Abschnitt diskutiere.

2.3 Surveillance Studies und die Rolle von Algorithmen

Eine neue Ebene der Wirkungsweise attestieren den Software-basierten Technologien auch die Forscher*innen der Surveillance Studies. Dabei stellen sie fest, dass es durch „neue“ Überwachungstechnologien zu einem erhöhten Verlust an Autonomie und Privatsphäre und einer stärkeren Diskriminierung Einzelner kommt (vgl. Schäufele 2017: 26 f.).

Marx (2002) verortet das „Neue“ der neuen Überwachung insbesondere in der Erweiterung der Informationsbasis durch digitale Speichermöglichkeiten und die Unsichtbarkeit der Datenerfassung. Dies führt aufgrund der Verfügbarkeit und geringen Kosten von großen Datenmengen in Kombination mit der automatischen Verarbeitung zu einer gesteigerten Ausweitung von Überwachungsmaßnahmen und insbesondere zur Verschiebung von der Betrachtung einzelner Verdächtiger auf große Mengen von Subjekten, die zunächst nicht verdächtig erscheinen müssen.

The new surveillance relative to traditional surveillance extends the senses and has low visibility or is invisible. It is more likely to be involuntary. Data collection is often integrated into routine activity. It is more likely to involve manipulation than direct coercion. Data collection is more likely to be automated involving machines rather than (or in addition to) involving humans. It is relatively inexpensive per unit of data collected. Data collection is often mediated through remote means rather than on scene and the data often resides with third parties. Data is available in real time and data collection can be continuous and offer information on the past, present and future (ala [sic!] statistical predictions). (ebd.: 15)

Demnach erleichtern die neuen Überwachungstechniken, insbesondere „immer größere Mengen an Informationen parallel aufzunehmen und zu verarbeiten und somit sehr große Gruppen an Individuen gleichzeitig zu überwachen“ (Schäufele 2017: 23).

Diese Art der Datenverarbeitung entspricht der Profilierung von Subjekten¹⁷ und führt zu einer Praxis des *Social Sorting* (vgl. Lyon 2003), also der Sortierung von Menschen auf Basis arbiträrer Merkmale. Die besondere Steigerung der Leistungsfähigkeit der Datenerfassung und Datenverarbeitung beruht auf den computerbasierten Möglichkeiten der effektiven Speicherung und automatisierten Verarbeitung dieser Informationen. Aufgrund der elementaren Rolle von Software in diesen Formen der Überwachungstechnik wird anstelle des *Social Sorting* auch von *Software Sorting* (vgl. Wood/Graham 2006) gesprochen. Dabei setzen Software-basierte Systeme auch bei „Rohdaten“ an, wie sie beispielsweise Videoüberwachungssysteme generieren, und reichern die vorhandenen Daten durch eine automatisierte Analyse mit zusätzlichen Informationen an. Ein weiterer Begriff lautet *Algorithmic Surveillance*:

17 Siehe auch Hildebrandt (2008c); eine ausführliche technik-soziologische Auseinandersetzung mit der Technik des Profiling findet sich in Schäufele (2017).

‘Algorithmic surveillance’ [...] is used specifically to refer to surveillance technologies that make use of computer systems to provide more than the raw data observed. This can range from systems that classify and store simple data, through more complex systems that compare the captured data to other data and provide matches, to systems that attempt to predict events based on the captured data. (Introna/Wood 2004: 18)

Ein wesentlicher Punkt bei dieser Art der Datenverarbeitung ist, dass die Algorithmen selbst in ihrer logischen Verarbeitungsstruktur in der Regel unsichtbar sind. Aber auch wenn der zugrunde liegende Quelltext betrachtet wird, ist nicht unbedingt klar, welcher Teil des Softwarecodes zur Laufzeit tatsächlich ausgeführt wird und welche Daten er zur gegebenen Zeit verarbeitet: “In short, software algorithms are operationally obscure“ (Introna 2007: 17).

Darüber hinaus sind aber die Algorithmen, die beispielsweise für Gesichtserkennung verwendet werden, häufig auch mathematisch derart komplex, dass nur wenige Experten sie nachvollziehen können. Offensichtlich sind diese aber manchmal selbst überrascht über die Ergebnisse, die diese Algorithmen produzieren. “[M]ost of the algorithms in facial recognition are based on very sophisticated statistical methods that only a handful of experts can interpret and understand. Indeed it seems that even they [Phillips u.a. 2003, Anm.d.Verf.] have been surprised by the behaviour of their algorithms” (Introna/Wood 2004: 183).

Tab. 2.1: *Opaque vs. transparent technology* (Introna 2007: 17)

Opaque technology is ...	Transparent technology is ...
Embedded / hidden	On the ‚surface‘ / conspicuous
Passive operation (limited user involvement, often automatic)	Active operation (fair user involvement, often manual)
Application flexibility (open ended)	Application stability (firm)
Obscure in its operation/outcome	Transparent in its operation / outcome
Mobile (software)	Located (hardware)

Introna und Wood fokussieren in ihrer Analyse von digitalen Überwachungstechniken und den *Politics of Information Technology* (vgl. Introna 2007) die Unterscheidung von *silent versus salient* beziehungsweise *opaque versus transparent technologies* (s. Tab. 2.1) – eine Unterscheidung, die insbesondere abhebt auf den unsichtbaren Charakter von Algorithmen, die häufig als Teil anderer Technologien eingebettet und passiv agieren. Dies impli-

ziert gewisse Einschränkungen in der Zugänglichkeit der Bedienung durch die User. Darüber hinaus stellt Intronas eine gewisse Flexibilität in der Arbeitsweise und den Ergebnissen fest. Demgegenüber skizziert er transparente Technologie als an der Oberfläche sichtbar und daher eher aktiv (mit User-Eingaben) arbeitend (Abschnitt 2.1).

Auch hier stehen also eine diagnostizierte Unsichtbarkeit und Unzulänglichkeit von Algorithmen und die Flexibilität der eingeschriebenen Logiken im Fokus der Kritik. Zudem bewirkt die empfundene Objektivität von Computern eine Verschleierung der in die Algorithmen eingeschriebenen Werte (vgl. Graham/Wood 2003: 232). Die kritische Wahrnehmung und Reflexion der Wirkung von Software tritt also auf verschiedene Weise zutage und von verschiedenen Seiten werden software-spezifische Eigenschaften problematisiert. Diese Charakterisierung von Software als spezifische Form von Technik möchte ich im nächsten Abschnitt aus technik-soziologischer Perspektive systematisieren – als Erklärungsansatz für die Kritik einer sonst in Technik weitgehend akzeptierten Experten-Laien-Differenz (Kap. 4).

3 Software als Element moderner Technologien

Wie im vorigen Kapitel beschrieben wird die Frage nach der Einflussnahme der Technik auf die Gesellschaft mit der wachsenden empirischen Bedeutung von Software zunehmend diskutiert. Die Problembeschreibungen der Literatur zu Techno-Regulation stellen die regulierenden Potenziale von Technik in anschaulichen Beispielen dar und die Critical Algorithm Studies fokussieren explizit die Rolle von Software. Nicht zuletzt kristallisiert sich auch in den Surveillance Studies Software als die Basis einer qualitativ neuen Art von Überwachungstechnologien heraus.

Zwar werden mitunter die Besonderheiten der auf Software basierenden Technologien in den Problembeschreibungen diskutiert, bislang fehlt jedoch eine theoretische Auseinandersetzung mit der Frage, was die Neuartigkeit darstellt. Während Gary T. Marx und Lucas Introna in ihren Charakterisierungen von „New Surveillance“ und „Opaque Technologies“ vor allem die Wirkung der Technik beschreiben, wie „integrated, automated, involving machines“ (Marx 2002) oder „hidden, obscure, flexible“ (Introna 2007), unternimmt Rammert (2006) den Versuch, stärker die technische Beschaffenheit autonom *erscheinender* Technologien zu identifizieren. Ausgehend von seiner Analyse leite ich spezifische Eigenheiten von Software ab. Die systematische Beschäftigung mit den Eigenschaften von Software dient dem Verständnis des Phänomens und schließlich der Beweggründe der erklärten Absicht der Freie-Software-Bewegung, Software als fundamentale Technologie *einsehbar und veränderbar* zu machen.

3.1 „Jenseits des mechanischen Bewirkens“

In seinem Aufsatz *Technik in Aktion* setzt sich Rammert (2006) mit der Frage auseinander, wie sich Technik vor dem Hintergrund der zunehmend interaktiven und autonomen Technologien bestimmen lässt.

Er argumentiert:

[...] die Aktivitäten bestimmter avancierter Techniken nicht mehr mit dem Vokabular des mechanischen Bewirkens und Befolgens angemessen begriffen werden können, sondern man zu Konzepten des Agierens und Interagierens greifen muss, will man den höheren Grad ihrer Autonomie beschreiben. (Rammert 2006: 165)

Vor dem Hintergrund der Feststellung, dass moderne Techniken „komplexer, kombinierter und undurchsichtiger geworden“ (ebd.: 169) sind – Wirkungen, die wie oben beschrieben Teil verschiedener wissenschaftlicher Diskussionen sind –, entwickelt er eine Reihe von *Dimensionen technischen Wandels* zur genaueren Analyse des Grades von Autonomie und Interaktivität von Technik, die auch in der Frage nach den spezifischen Eigenheiten von Software nützliche Kategorien darstellen.

Unter *Komplexität* betrachtet Rammert die „Menge der Elemente und ihrer Beziehungen untereinander“. Technik besteht demnach also zunehmend aus verschiedenen Komponenten, die miteinander in Beziehung stehen. *Kombiniertheit* beschreibt dabei die „Integration heterogener Techniken in einem technischen System“, also beispielsweise die Kombination von Sensortechniken mit Speichertechniken und Regelungstechniken.

Ein weiteres Merkmal bildet die *Undurchsichtigkeit*, die wiederum zusammenhängt mit der „Vielzahl von Elementen und Beziehungen und auch mit der Unterschiedlichkeit der kombinierten Typen“.

In diesem Zusammenhang stellt Rammert fest, dass insbesondere Computer und „mit Rechneinheiten kombinierte Techniken“ durch die Programmierbarkeit der Abläufe hier eine besondere Rolle spielen, da sie „im Hinblick auf ihre Funktion und den zu erwartenden Ablauf“ nicht mehr so leicht einzusehen sind. Software besteht selbst aus verschiedenen Modulen, die über Schnittstellen miteinander kommunizieren, und über Software können unterschiedliche heterogene Elemente wie Sensoren, mechanische Sachtechniken oder alltägliche Gegenstände miteinander verbunden werden. Die Kombination der technischen Elemente ist dabei nicht unbedingt unmittelbar sichtbar, ebenso wenig die Beziehung, in der die verschiedenen Geräte zueinander stehen.

Als Abgrenzung zum autonom *erscheinenden* Roboter und dem virtuellen Objekt führt Rammert hierbei einen fahrbaren Kran oder eine Tennisball-Wurfmaschine an, die weniger Freiheitsgrade bezüglich ihrer nächsten Bewegung haben und daher von außen als leichter zu berechnen und ihren Handlungsabläufen weniger kontingent – und somit weniger autonom – erscheinen. Dies ändert sich, sobald in den fahrbaren Kran oder die Tennisball-

Wurfmaschine eine Software-Logik eingebaut wird, die Freiheitsgrade in die Technik einbettet und sie programmierbar macht.

Gegenüber diesen *sachlichen* Merkmalen ergänzt Rammert *räumliche* Merkmale, darunter den Grad der *Globalisierung* – ein zunächst sperrig wirkender Begriff, der sich aber nicht nur auf die Verbreitung von Technologien bezieht, sondern auf die delokalisierte Vernetzung von technischen Objekten. Diese Vernetzung von Technik oder technischen Elementen erstreckt sich durch weltweite Telekommunikationsnetze über den ganzen Globus, insbesondere in der Internet-basierten Kommunikation spielt die globale Dimension eine zentrale Rolle.

Darüber hinaus beschreibt Rammert *Mobilität* als Merkmal moderner Technik. Darunter versteht er vor allem die „autonome Mobilität“ technischer Elemente, die durch die „Kombination der Bewegungstechnik mit der Sensor-, der Nachrichten-, der Übertragungs- und der Rechentechnik“ (ebd.: 170) aufkommt.

Als *zeitliches* Merkmal betrachtet er die *Variabilität* von Technik. Hierin besteht der größte Sprung bezüglich der vorbestimmten Abläufe von Technik. Hier spielt die rasante Entwicklung der Speichertechniken eine zentrale Rolle, da dadurch immer längere Abschnitte „der Vergangenheit und ihrer Varianten für die Gestaltung der zukünftigen Abläufe herangezogen werden“ können. Im Gegensatz zu „konventioneller Technik“ sind so die maschinellen Abläufe nicht mehr „geschrumpft auf die Repetition des Arbeitsvorgangs“ (ebd.: 170), vielmehr können Maschinen verschiedenste Informationen über ihre eigene Funktionalität bis hin zu externen Zusammenhängen in Form von gespeicherten Daten erfassen. Eine Berechnung dieser Daten ergibt dann eine Art „Erfahrung“, an die zukünftige Abläufe angepasst werden können.

Rammert resümiert, dass dadurch der Eindruck einer relativen Autonomie entsteht, obwohl es sich grundsätzlich noch um die „programmierte Kombination determinierter Systeme handelt“ (ebd.). Die Determiniertheit eines selbst lernenden Algorithmus ist allerdings nur sehr abstrakt fassbar und die Einflussnahme menschlicher Handlungen auf den Algorithmus wirkt vor dem Hintergrund der geringen Nachvollziehbarkeit der auf komplexen Berechnungen basierenden Entscheidungen der Softwarelogik marginal. Umso wichtiger ist es, sich die soziale Konstruktion der programmierten Logiken bewusst zu machen und die dadurch eingebauten Determinanten zu reflektieren – insbesondere, da sich die unvorhersehbaren Effekte durch die autonome

Weiterentwicklung von selbstlernenden Algorithmen, basierend auf zunächst unbekanntem zukünftigen Eingaben, potenzieren.

Zunehmend agieren agentenbasierte Software-Programme in relativer Autonomie, reaktiv und orientiert an Tätigkeiten, beziehungsweise bezogen auf andere *Agenten*. Durch derartige Verschachtelungen und offene Verknüpfungen zwischen Algorithmen entstehen „Aktionsräume mit höheren Freiheitsgraden“, wenngleich „Computerprogramme per definitionem weiterhin Algorithmen“ (ebd.: 171) bleiben.

3.2 Spezifische Merkmale von Software

Rammert nennt hier einige Merkmale, die in besonderer Weise bei Software-basierter Technik zum Tragen kommen oder in dieser Art von Technik besonders ausgeprägt sind. Zum einen führt er aus, dass Maschinen durch die einprogrammierten Algorithmen bezüglich ihrer Funktion und dem zu erwartenden Ablauf nicht mehr so leicht einzusehen sind und sich so durch die Kombination von *Komplexität* und *Kombiniertheit* verschiedener Module und Systeme ein besonderes Maß an *Undurchsichtigkeit* ergibt.

Die Programmierung ist hierbei nicht notwendigerweise statisch, wie etwa bei verdrahteten oder verlöteten Maschinen, sondern lässt sich nachträglich verändern. Die Möglichkeit der Speicherung von Informationen (*Variabilität*) und vor allem die Verwendung der gespeicherten Informationen in der Logik des Algorithmus führen zu den genannten erhöhten Freiheitsgraden. In Verbindung mit der ortsunabhängigen Vernetzung von Maschinen wird es sehr schwer nachzuvollziehen, welche Geräte miteinander in welcher Beziehung stehen und wie sich welche dieser Beziehungen auf den aktuellen Ablauf der Maschine auswirken. Die mit Computer verbundenen Systeme erreichen somit einen hohen Grad von Interaktivität bis hin zu autonom (erscheinenden) Prozessen.¹⁸

¹⁸ Rammert schließt an seine Entwicklung der ausgeführten Merkmale technischen Wandels eine Aktivitätsskala von Technologien an, die von *passiven* Werkzeugen wie etwa einem Hammer, über *aktive* und *reaktive*, also anpassungsfähigen Typen, bis hin zu *interaktiven* Technologien unterscheidet, die technische Systeme beinhalten, die sich aufeinander beziehen. Schließlich führt er *transaktive* Technologien an, die aber noch *Konstellationen von Mensch und Technik* vorbehalten bleiben. In dieser Typisie-

Es zeigt sich, dass die genannten Eigenschaften wie die Vernetzung verschiedener Technik-Elemente, die Speichermöglichkeiten und die Möglichkeiten einer reflexiven Informierung des Programms durch gespeicherte Daten insbesondere in Verbindung mit Software stehen. Sie bilden die wesentlichen Fokuspunkte der oben geschilderten Problembeschreibungen der Techno-Regulation, Algorithm Studies und Surveillance Studies (Kap. 2). Nicht zuletzt dreht sich ein Teil der Problembeschreibung ebenfalls um die zunehmende Bedeutung der Software, indem sie eingebettet in allerlei andere vormals analoge Technologien erscheint und somit ihr Potenzial und die damit verbundenen Kritikpunkte zum Teil eines *generellen technischen Wandels* werden.

Mit Ausnahme der Kategorie Mobilität, die ich in der Tabelle ausgespart habe, erweisen sich *Rammerts Merkmale technischen Wandels* als Eigenschaften, die bei Software in einem deutlich höheren Grade auftreten als in anderer Technik. In Tabelle 3.1 fasse ich daher die Merkmale zusammen, die mir für die Charakterisierung von Software wesentlich erscheinen.

Tab. 3.1: *Software als zentrales Element technischen Wandels, nach Rammert (2006: 169f.)*

<i>Komplexität</i>	Menge und Relationen der involvierten Elemente
<i>Kombiniertheit</i>	Integration heterogener Elemente
<i>Undurchsichtigkeit</i>	Folge von Komplexität und Kombiniertheit, zusätzlich eingebettete Funktionalität der Software
<i>delokalisierte Vernetzung</i> („Globalisierung“)	vernetzte Operationen der Elemente
<i>Variabilität</i>	Speichermöglichkeiten als Input

Diese Eigenschaften spiegeln sich wider in den oben zusammengefassten Problembeschreibungen der Literatur, wie beispielsweise den eingebetteten passiven und versteckten Operationen (Tab. 2.1) (vgl. Introna 2007), die Rammert als *Kombiniertheit, Komplexität und Undurchsichtigkeit* auf die Integration heterogener Elemente und deren Relationen zurückführt. Diese bei Software besonders stark in Erscheinung tretenden Eigenschaften treten

ung schreibt Rammert neben den Dimensionen *Motorik* (Antrieb), *Aktorik* (zunehmend automatisierte Abläufe) und *Sensorik* (Erfassung der Umwelt) übrigens der *Informatik* eine tragende Rolle zu, sie bildet den Übergang von „der festen Regelbefolgung zu einer flexibleren Programmierung“.

auf in Kombination mit *delokalisierter Vernetzung* und *Variabilität*. Der Begriff *delokalisierte Vernetzung* ersetzt in meiner Fokussierung auf Software den Begriff Globalisierung, da die globale Umspannung der technischen Netze sehr wohl eine wichtige Feststellung beinhaltet, aber darauf beruht, dass technische Komponenten – über die Kombination innerhalb eines lokalen *Maschinengehäuses* hinaus – lose Koppelungen mit entfernten Einheiten aufnehmen können. Zudem impliziert der Begriff Globalisierung hier zu starke Konnotationen sozio-ökonomischer Zusammenhänge, die hier aber ausgeblendet bleiben. *Variabilität* im Sinne erweiterter Speichermöglichkeiten ermöglicht über die kurzfristige Reaktion auf die Umwelt hinaus langfristige Veränderungen des Algorithmus selbst. Dies geschieht über die Aggregation von Daten und die Erkennung von Mustern und schließlich der Ableitung prospektiver Annahmen, basierend auf reichhaltigen Datenreihen.

Die hier ausgearbeiteten spezifischen Besonderheiten von Software begründen und motivieren die Auseinandersetzung mit der Frage nach der Legitimation einer exklusiven Softwareentwicklung (die die Nutzer von der Entwicklung ausschließt), wie sie die Free Software Foundation kritisiert, und legt den theoretischen Grundstein einer Diskussion der Dichotomie zwischen Technik herstellenden Expert*innen und darauf in der Nutzung zurückgreifenden Laien. Diese Diskussion erfolgt im nächsten Kapitel.

Teil II

Epistemische Regime zur Regulierung der Wissensdifferenz

4 Technik als soziales Verhältnis zwischen Expert*innen und Laien

Nachdem ich nun spezifische Eigenschaften der Technik *Software* herausgearbeitet und die Problematisierungen Software-basierter Technik dargelegt habe, die offensichtlich in Verbindung zu diesen Eigenschaften stehen, werde ich die grundsätzliche Wissensdifferenz zwischen *Technik-Hersteller*innen* und *Technik-Nutzer*innenn* als *Expert*innen* und *Laien* technik-soziologisch anhand der *Sozialtheorie der Technik* von Ingo Schulz-Schaeffer (2000) diskutieren. Vor dem Hintergrund der beschriebenen Kritik der Legitimität einer exklusiven Software-Herstellung, die Nutzer*innen den Einblick in und die Veränderung von Quelltext verwehrt (Abschnitt 1.4), konzentriere ich mich hierbei auf die Problematisierung der *Macht einer geschlossenen Expertene-lite* in der Gestaltung von Technik und der kritischen Rolle von *Vertrauen der Laien in Technik* und ihre erwarteten Wirkungen.

4.1 Technik als stabiler Wirkungszusammenhang

Üblicherweise zeichnet sich Technik dadurch aus, dass sie bestimmte Funktionsprinzipien regelhaft definiert und somit ihren Nutzer*innen gewisse Funktionen als Wirkungszusammenhänge zur Verfügung stellt. Nutzer*innen benutzen die Technik dabei, ohne die Funktionsprinzipien notwendigerweise zu verstehen. Vielmehr wird die Technik von *Expert*innen* entwickelt, die ein spezifisches Wissen über die Funktionszusammenhänge besitzen. Die Entwicklung der Technik geschieht dadurch *entbettet* aus sozialen Zusammenhängen, sie wird also zunächst unabhängig vom situativen Kontext der Akteure festgeschrieben.

Die Wirkungsweise von Technik ist definiert durch festgelegte, *formulierte* Regeln. Im Gegensatz zu praktischen Regeln sozialer Strukturen, die nicht oder nur *ex post* formuliert werden, um die etablierte Praxis sinnhaft zu beschreiben, legen sie also gesicherte, stabile Ereigniszusammenhänge fest. Folglich ergibt sich eine besondere Wirksamkeit aus den darin angelegten (formalen) Strukturen, die unabhängig von alltäglichen Zusammenhängen

eine wiederholbare Wirkung entfalten, auf die als Ressource des Handelns zugegriffen werden kann (vgl. Schulz-Schaeffer 2000: 17).

In der Benutzung der Technik interpretieren die Nutzer*innen die in die Technik eingeschriebenen Regeln und entwickeln darauf basierend Routinen. Diese sind nicht selbst Teil der formulierten Regeln, sondern entstehen durch die praktische Bezugnahme auf die Technik.

Die Nutzer*innen haben in ihrer Praxis der Nutzung auch gewisse Interpretationsspielräume, *interpretative Flexibilität* (vgl. Kline/Pinch 1996). Ein Mikrowellengerät kann zum Beispiel auch verwendet werden, um RFID-Chips zu zerstören, dies ist aber nicht unbedingt von den Hersteller*innen vorgesehen, stellt also eine kreative Interpretation der zur Verfügung gestellten Funktion dar. Ungeachtet dessen stellt das Gerät einen klaren Wirkungszusammenhang bezüglich zu erwärmender Speisen dar, ohne dass ein Nutzer die Funktionsweise von Mikrowellen verstehen müsste.

Der Wirkungszusammenhang, der durch die Mikrowelle anwendbar wird, stellt aber einen *gesicherten Ereigniszusammenhang* zwischen dem Bedienenhandeln und der technischen Funktion dar, auf die die User ohne Kenntnis der zugrunde liegenden Prinzipien zugreifen können. Dies führt auf Seiten der Techniknutzer*innen zu einer Handlungsentlastung und schließlich einer Leistungssteigerung durch die einfache Verfügbarkeit dieses Wirkungszusammenhangs als Ressource (vgl. Schulz-Schaeffer 2000: 18). Dieser Zusammenhang zwischen Handlungsentlastung und Leistungssteigerung macht für Blumenberg (1981: 7) das Wesen des Technischen aus (zit. in Schulz-Schaeffer 2000: 18). Wie in den vorigen Kapiteln klar wurde, sind die Wirkungszusammenhänge bei Software aber nicht unbedingt immer so sicher und gewiss, wie erwartet, was das Vertrauen in die Technik mindert. Dies ist Gegenstand des nächsten Abschnitts.

4.2 Vertrauen in Technik

Die Laien sind zunächst entlastet vom Verständnis der Funktionsweise der Technik und dem damit verbundenen Expertenwissen, das lediglich die Expert*innen benötigen, die die Technik herstellen. Da die Wirksamkeit der Regeln auf Prinzipien beruht, die die Anwender*innen, Laien, nicht verste-

hen müssen, besteht folglich eine *Wissensdifferenz* zwischen Expert*innen und Laien.

Expertensysteme generell beruhen auf wissenschaftlichem Wissen.¹⁹ In Technik ergibt sich aus der Wissensdifferenz zwischen Expert*innen und Laien die Notwendigkeit des Vertrauens in die Richtigkeit dieses Wissens. Aufgrund der nicht immer niederschweligen Zugänglichkeit und der Komplexität der Materie gestaltet sich die Nachvollziehbarkeit wissenschaftlichen Wissens, wie dieses zustande kommt und auf welchen Prämissen es beruht, für Laien mitunter schwierig; dadurch reduziert sich die Möglichkeit, dessen Validität und Legitimität zu überprüfen, erheblich.

Stärker noch gilt dies für Technik, deren Funktionsprinzipien häufig auch materiell verkapselt und somit nicht mehr sichtbar und nachvollziehbar sind. Für die Nutzung von Technik benötigen die Laien-Nutzer*innen prinzipiell lediglich eine gewisse Kenntnis der Benutzungsweise, die mit den darunter liegenden Funktionsprinzipien nicht notwendigerweise viel zu tun haben muss. Dieses *Benutzungswissen* gibt dem Laien gewisse Regeln an die Hand, um die gewünschte Wirkung der Technik zu erreichen und für sich nutzbar zu machen (vgl. Schulz-Schaeffer 2000: 218 ff.).

Es gibt verschiedene Strategien, Expertenwissen in Benutzungswissen zu übersetzen (vgl. ebd.: 360 ff.):

1. Die Nutzer*innen bekommen eine grundlegende Schulung – beispielsweise wird die Erlaubnis des Führens eines Kraftfahrzeugs an eine absolvierte Fahrschule gekoppelt. Auch manche Software ist nur mit einer ausgiebigen Schulung sinnvoll nutzbar.
2. Schaffung von Zugangspunkten zu Benutzungswissen – beispielsweise über Artefakte wie Gebrauchsanweisungen oder Wikis oder aber über entsprechende Verweise, sich an Expert*innen zu wenden (z.B. „Zu Risiken und Nebenwirkungen ...“, oder „... wenden sie sich an ihren Systemadministrator“)
3. Verlagerung der Benutzung in Expertenhandeln – beim Fliegen wird das Führen des Flugzeugs an Expert*innen, den Piloten oder die Pilotin, ausgekoppelt. Mehr noch, die „Nutzer*innen“ des Flugzeugs werden im Grunde vom Eingang des Startflughafens bis zum Ausgang des Zielflug-

19 Zu Giddens' Begriff von Expertensystemen – im Unterschied zur Begriffsbestimmung als Teilgebiet der ‚Künstlichen Intelligenz‘ (KI) – und der Abstützung von Expertenwissen auf wissenschaftlichen Prinzipien wie Nachvollziehbarkeit und Kritik vgl. Giddens (1996).

hafens von geschultem Personal geführt. In Organisationen ist die Administration von Computern in aller Regel auch den Expert*innen der IT-Abteilung überlassen, an die sich die Nutzer*innen idealiter vertrauensvoll wenden können.

4. Selbstverständlichung der Handlungspraxis der Benutzung – dies kann zu weiterer Unsichtbarmachung der zugrundeliegenden Regeln führen, weil durch die Selbstverständlichkeit die Kontingenz der Regeln ebenfalls unsichtbar wird. Der richtige Umgang im Straßenverkehr etwa ist ein komplexer Vorgang, der von klein auf antrainiert wird, damit er in „Fleisch und Blut“ übergeht. Dass man DOS-Computer einfach ausschalten konnte und das „Herunterfahren“ in den 90ern ein Novum war, kann man sich heute kaum noch vorstellen. Auf ähnliche Weise hielten sich „Drop-down-Menüs“ bis zum Aufkommen von Touch-Geräten und der „Start-Button“ von MS Windows galt lange als selbstverständliche Gegebenheit.

Durch die aus dem unmittelbaren Kontext der Benutzung entbettete Entwicklung der Technik und das fehlende Verständnis der Funktionsweise der Technik beruht deren Benutzung auf einem Vertrauen in die Gesicherheit der Ereigniszusammenhänge und der Korrektheit der Funktionsweise (vgl. ebd.: 215) beziehungsweise in einem Vertrauen in die Regeln der Benutzung. Das Vertrauen in die Regeln impliziert, dass Laien davon ausgehen müssen, dass die Regeln, die für die Benutzung angegeben werden, auch die gezielten Effekte bewirken und keine anderen (vgl. ebd.: 356).

Wie im vorigen Kapitel beschrieben, sind die Wirkungszusammenhänge der Funktionsweise von Software, durch ihre Eigenschaften Variabilität und Flexibilität, nicht immer sicher und stabil (Abschnitt 3.2). Zwischen Eingabe und Ausgabe werden Daten generiert, ausgetauscht und in den Programmverlauf integriert – Daten, die teils räumlich entfernt sind und damit weder sichtbar noch bekannt.

Sind generell die Regeln der Technik, durch ihre Entbettung im Zuge der Differenzierung Experte/Laie, undurchsichtig oder hinter der Technik verborgen (vgl. ebd.: 225), so steigert sich diese Unsichtbarkeit durch die entfernten Abläufe und die visuelle Entkoppelung der Oberfläche infolge ihrer Virtualisierung erheblich (Abschnitt 3.1). Der Zusammenhang zwischen Benutzungspraktiken und Funktionsregeln ist dadurch in besonderem Maße verschleiert, wodurch es erschwert wird, die Wirkungsweise nachzuvollziehen. Dadurch sind User darauf angewiesen, induktive Schlüsse zu ziehen. Mit anderen Worten, die „Gesicherheit“ der Ereigniszusammenhänge (zwischen Handeln und Ergebnis der Handlung) ist insofern gehemmt, als regel-

mäßig Ereignisse stattfinden können, die für die Anwender*innen nicht ersichtlich sind oder nicht antizipiert wurden.

Beispielsweise kann das Ergebnis einer Suche im Internet nach denselben Begriffen unterschiedlich sein in Abhängigkeit von Zeit und Ort und Merkmalen des Computers, da der Algorithmus abhängig von vielen Faktoren Suchpräferenzen der Nutzer*innen antizipiert und diese in die Bewertung der Suchergebnisse einbezieht. Oder wenn ein Programm für einen Moment keine Rückmeldung gibt, ist es stets (und theoretisch beweisbar)²⁰ ungewiss, ob dies nur für einen Moment gilt, da das Programm gerade den nächsten Schritt berechnet, oder ob das Programm aufgrund einer Eingabe in einer Endlosschleife hängt beziehungsweise „abgestürzt“ ist. Das Vertrauen ist also durch die Unsichtbarkeit der wirksamen Abläufe in Verbindung mit der Unsicherheit über die tatsächlichen Wirkungen der Bedienung gestört.

Ähnlich beschreibt Weick (1990) in seinem Aufsatz *Technology as Equivoque* eine veränderte Wahrnehmung der User von Technik infolge einer zunehmenden „dynamisch interaktiven Komplexität“. Für ihn charakterisieren sich die „neuen Technologien“ durch

- *stochastisch* erscheinende Ereignisse, deren Ursache-Wirkungs-Zusammenhänge unbekannt sind,
- *kontinuierliche* Ereignisse, also technische Ablaufzusammenhänge, deren einzelne Abläufe direkt miteinander interagieren, sodass ein Ereignis im technischen System weit entfernte Wirkungen zeitigen kann, sowie
- *abstrakte* Ereignisse, d.h. eine Verbergung der internen Struktur sachtechnischer Artefakte (vgl. Schulz-Schaeffer 2000: 349).

Weick versteht unter stochastischen Ereignissen solche, die *scheinbar* zufällig auftreten – scheinbar, weil es sich dennoch um deterministische Maschinen handelt, die darunter liegenden Prozesse aber von den Usern nicht hinreichend verstanden werden und sich das Design der Prozesse häufig ändert. So treten für die User wichtige Ereignisse zufällig und unvorhersehbar auf, und zwar so selten, dass User nicht die Zusammenhänge verstehen, aber so häufig, dass sie irritieren (vgl. Weick 1990: 10).

Für die zunehmende Relevanz dieser Eigenschaften „neuer Technologien“ schreibt Weick Computern eine besondere Rolle zu, weil diese häufig

20 Das Halteproblem, ein Theorem der Informatik, besagt, dass es keinen Algorithmus geben kann, der die Frage, ob ein Programm anhält (bis zum Endzustand durchläuft), für alle möglichen Algorithmen und beliebige Eingaben entscheiden kann (vgl. Davis 1985: 70f.).

keine vollständige und akkurate Indikation des aktuellen Prozesses geben und sich generell die Status-Anzeigen und die Kontrollaktivitäten vom tatsächlichen Prozess-Zustand entkoppeln (vgl. ebd.: 8). Dies verstärkt in der Folge die schon von Perrow (1987) thematisierten Probleme interaktiver Komplexität, einer Kombination enger Koppelung von Betriebsabläufen und starker technischer Komplexität. Die Wahrnehmungs-Effekte dieser kontinuierlichen Ereignisse werden durch das vermehrte Auftreten von scheinbar zufälligen Ereignissen zusätzlich verstärkt, die zur ohnehin schon hohen Komplexität die Nachvollziehbarkeit der Ursache-Wirkungs-Zusammenhänge zusätzlich erschweren.

Im Grunde geht auch die dritte Kategorie der *abstrakten* Ereignisse in dieselbe Richtung: Die Verbergung der inneren Struktur und Abläufe führt zu einer Entkopplung der Wahrnehmung der Ausführungszustände einer Maschine und der tatsächlichen Abläufe.

Weick argumentiert, dass durch die Steuerungsintervention des Users letztlich ein neues technisches System generiert wird, das weder vom User verstanden wird noch von den Selbststeuerungs-Elementen valide interpretiert werden kann: "New technologies are technologies involving a technology in the head and a technology on the floor" (Weick 1990: 17).

Interessant ist hierbei die sich in der zentralen Rolle der „mentalen Representationen“ (ebd.: 39) andeutende Virtualisierung der Technik, die sich in Form des Graphical User Interface offenbart. Wie in Abschnitt 1.1 ausgeführt, besteht eine strukturelle Ähnlichkeit der Kommandozeile und der Logik der Programmbefehle der Software. Während die manuellen Befehls-eingaben also ihre Entsprechung in der Softwarelogik finden, bildet die grafische Oberfläche eine neue Qualität von Abstraktion. Dies findet sich wieder in den Beschreibungen von MS-DOS-Usern, die ein Gefühl eines Kontrollverlustes verspüren in der Konfrontation mit der grafischen Oberfläche eines Apple-Computers (Turkle 1995: 39). Auch Chun (2011: 95) formuliert die Flüchtigkeit digitaler Oberflächen: "Digital media is not always there (accessible), even when it is (somewhere). We suffer daily frustrations with digital 'sources' that just disappear. Digital media is degenerative, forgetful, erasable."

Diese Doppelstruktur zwischen Bedienelement und Funktionslogik von Technik ist zwar schon in einem einfachen Kipp-Schalter angelegt, erreicht aber durch die starke Entkopplung von Interface und Logik (Weick) und die zunehmende technische Autonomie durch Undurchsichtigkeit, Vernetzung und Variabilität (Rammert) eine neue Qualität (s. voriges Kap.).

Zwar ist die dadurch gesteigerte Undurchsichtigkeit und Interaktivität nicht allein Software-basierten Technologien vorbehalten – denn grundsätzlich kann jede Software-Logik auch in Hardware-Schaltungen kodiert werden, aber Software ist demgegenüber nicht „hart“ kodiert und damit wörtlich in (Platinen-) Form gegossen, sondern kann durch ihren nicht materiellen Charakter als zeichenbasiertes Medium flexibel verändert werden.

Das Vertrauen der Nutzer*innen ist bei Software-Technik also potenziell gestört, was die Beziehung von Hersteller*innen als Expert*innen und Laien als Nutzer*innen potenziell stört und die Legitimation und Nützlichkeit dieser kategorischen Dichotomie infrage stellt. Ein weiterer Aspekt dieser Legitimitätsfrage berührt die Frage der gestalterischen Macht der Expert*innen.

4.3 Die Macht der Expert*innen

Durch die Entbettung der Regelformulierung aus ihrem Nutzungskontext ergibt sich neben der Wissensdifferenz auch eine gewisse Machtasymmetrie. Die Handelnden befolgen in der Nutzung der Technik Regeln, die von Expert*innen ausgearbeitet wurden und die sie als Laien weder in ihrer Wirkungsweise verstehen noch einen Einfluss auf ihre Gestaltung haben: Sie können sie lediglich als Ressourcen verwenden. Mit der genannten Entlastung der Laien geht also ein gewisser *Kontrollverlust* einher.

Diese „Macht“ der Experten wird prinzipiell durch zwei Aspekte relativiert. Erstens können die Expert*innen nicht außer Acht lassen, dass die Technik wieder in Handlungsabläufe eingebunden wird, und müssen diese daher anschlussfähig gestalten. Sie treffen Annahmen über zukünftig erwartete oder benötigte Praktiken der Nutzung der bereitgestellten Ressourcen – andernfalls findet die produzierte Technik möglicherweise keine Nutzer*innen.

Zweitens veräußern die Expert*innen durch die Bereitstellung eines Technikprodukts die darin materialisierten Wirkungszusammenhänge und haben keinen weiteren Einfluß auf deren Verwendung. Damit können sie zunächst von ihrem Wissensvorsprung nicht mehr profitieren, da die Nutzer*innen das in Technik vergegenständlichte Wissen für sich nutzbar machen können (vgl. Schulz-Schaeffer 2000: 324 ff.).

Allerdings führen die monopolartigen Marktstrukturen in der Welt der Software und der Software-basierten Services dazu, dass Hersteller nur in einem sehr begrenzten Maße darauf angewiesen sind, auf die Befindlichkeiten ihrer Nutzer*innen Rücksicht zu nehmen. Vielmehr setzen sie durch ihre Hegemonie Standards, die aufgrund von Pfadabhängigkeiten und mangelnden Alternativen (oder mangelnder Kenntnis von Alternativen) von den Nutzer*innen angenommen werden *müssen*. Beziehungsweise führen die Pfadabhängigkeiten dazu, dass Alternativen sich faktisch nicht durchsetzen können gegen einen etablierten Standard. Ein faszinierendes Beispiel für die Etablierung eines Quasi-Monopols durch strategischen Wettbewerb stellt hierbei die Entwicklung von Microsoft dar. Durch einen Deal mit IBM erreichte Microsoft, dass auf allen IBM-Systemen ihre Variante des Disk Operating Systems DOS ausgeliefert wurde, und legte damit den Grundstein für ihre Marktmacht bis heute. Heute bildet neben MS Windows auch deren Office-Paket einen etablierten Standard, dessen Hegemonie allein aufgrund seiner ubiquitären Nutzung kaum anfechtbar ist.²¹

Entgegen der zweiten Abschwächungsthese steht die Eigenschaft von Software, dass die Art und Weise der Verwendung der Software eben nicht notwendigerweise außerhalb des Einflussbereichs der Hersteller liegt. Vielmehr lassen sich durch die Implementierung von Datenströmen die Nutzungsweisen detailliert erfassen und durch den Fernzugriff über Hintertüren oder über reguläre Software- (oder Firmware-) Updates die Funktionalitäten im Nachhinein ändern und damit die Verwendung einschränken. Auch hier spielen also die Softwareeigenschaften *Variabilität* und darüber hinaus *delokalisierte Vernetzung* eine zentrale Rolle (vgl. Tab. 3.1) für das Machtverhältnis zwischen Hersteller-Experte/in und Laien-Nutzer*in.

Ein prominentes Beispiel hierfür ist der Fall der Spielekonsole PlayStation 3, die die Möglichkeit bot, ein Linux-System zu installieren und die ansonsten im Funktionsumfang sehr beschränkte Konsole um die Möglichkeiten eines Heim-PC zu erweitern. Über ein nachträgliches Firmware-Update wurde die Funktionalität verändert, sodass kein Linux mehr installiert werden konnte (vgl. Kushner 2012).

21 Aus Platzgründen kann ich diesen faszinierenden Wirtschaftskrimi leider nicht nach erzählen, empfehle aber wärmstens die Lektüre von Grassmuck (2004: 204ff.). Über die Rolle von Pfadabhängigkeiten in den Entscheidungen und Abwägungen über Alternativsysteme siehe Dobusch (2008).

Software als Kontrolltechnologie

Wie in den vorigen Kapiteln erörtert, spielt Macht auch eine besondere Rolle in Bezug auf das Kontrollpotenzial von Software im Sinne von Überwachung, also auch im Sinne von Steuerung. Hierbei spielen die in Kapitel 3 erarbeiteten Eigenschaften von Software zusammen:

- Undurchsichtigkeit,
- die delokalisierte Vernetzung sowie
- die Variabilität aufgrund der unermesslich gewordenen Speichermöglichkeiten.

Diese Eigenschaften (vgl. auch Tab. 3.1) spiegeln sich wider in den Diskursen der regulativen Wirkung von Technik, der Steuerpotenziale von Algorithmen und der Algorithmic Surveillance (s. Kap. 2), die letztlich alle die Macht von Technik im Allgemeinen problematisieren und in besonderer Weise die verstärkenden Wirkungen von Software im Speziellen problematisieren.

Der Speicherung von Informationen („Variabilität“) schreibt auch Giddens (1984: 33f.) eine besondere Rolle für die Ausübung von Kontrolle zu. Er diskutiert die Bedeutung von Ressourcen für die Herrschaft über materielle Phänomene, Güter und Umwelt (*allokative Ressourcen*) und die Herrschaft über Personen (*autoritative Ressourcen*). Über die Speicherung von Informationen können demnach soziale Beziehungen über ein längeres Raum-Zeit-Kontinuum hinweg geregelt werden.

Das heißt, zur Organisation (im Sinne des prozesshaften Organisierens) von Menschen werden Informationen und Kenntnisse benötigt. Mit deren Speicherung über die Begrenztheit des menschlichen Gedächtnisses hinaus ergibt sich eine wesentliche Erweiterung möglicher Machtausübung (vgl. Schulz-Schaeffer 2000: 168ff.). Vor dem Hintergrund der effizienten Speicherung, Bearbeitung und Bereitstellung von Informationen durch Software zeichnet sich auch in der Argumentation von Giddens das Potenzial des Computers zur Kontrollarchitektur ab.

Aktualisierung der Kritik der Free Software Foundation

Während Richard Stallmans Kritik eher auf die Möglichkeit der Aneignung von Technik abzielte (der Stein des Anstoßes war laut einer Anekdote ursprünglich die eigene Modifikation eines Drucker-Treibers), erfolgte 2013 eine Aktualisierung der Forderung nach Freie-Software-Lizenzen infolge der globalen Bewusstwerdung der *tatsächlich* ausgeübten (und nicht lediglich

befürchteten) staatlichen Kontrolle von Software-Usern (also allen Bürger*innen) durch Hintertüren in verbreiteten Software-Anwendungen. Vor dem Hintergrund der Leaks von Edward Snowden im Juni 2013 hebt Stallman (2013) unter dem Titel „Why Free Software Is More Important Now Than Ever Before“ die Möglichkeit hervor, durch Software die User zu überwachen: “In extreme cases (though this extreme has become widespread) proprietary programs are designed to spy on the users, restrict them, censor them, and abuse them” (ebd.). Ferner bezeichnet die Free Software Foundation die fehlenden Einflussmöglichkeiten auf die Gestaltung von Software als ein Instrument ungerechter Machtverteilung:

We campaign for these freedoms because everyone deserves them. With these freedoms, the users (both individually and collectively) control the program and what it does for them. When users don't control the program, we call it a “non-free” or “proprietary” program. The nonfree program controls the users, and the developer controls the program; this makes the program an instrument of unjust power. (Free Software Foundation 2013)

Diese Kritik stellt die Legitimität der Software-Entwickler als einer privilegierten Klasse mit exklusiven Informations- und Modifikationsrechten bezüglich der von ihnen verarbeiteten Software infrage. Dies steht im Zusammenhang mit den diskutierten spezifischen Eigenschaften von Software: Als *Silent Technology* (vgl. Introna/Wood 2004) operieren Informations-Technologien häufig im Hintergrund als unsichtbare Agenten, können aber aufgrund der Fähigkeit, in kurzer Zeit große Mengen von Daten zu verarbeiten, leicht zur Kontrolle von Menschen und Gütern verwendet werden.

Auf die im Hintergrund laufenden Prozesse, die unbemerkt User-Daten an Dritte senden, zielt auch die aktuelle Kritik von Richard Stallman. Die potenzielle *Kontrolle* hat also verschiedene Dimensionen: von Kontrolle im Sinne von Überwachung oder Monitoring (Graham und Wood 2003) über die mehr oder weniger subtile Lenkung ihrer Aufmerksamkeit über Filter-Bubbles (vgl. Pariser 2011) bis hin zur direkten Beeinflussung ihrer Handlungsweisen über in Technik eingeschriebene Normen (vgl. Latour 1991).

Ein verändertes Experten-Laien-Verhältnis

Wie ausgeführt wurde, kann zwar eine Trennung von Expert*innen und Laien als ein Merkmal von Technik betrachtet werden. Die dadurch entstehende Wissensdifferenz zwischen Hersteller*innen und Nutzer*innen bringt aber mit sich, dass Expert*innen eine gewisse Gestaltungsmacht besitzen und

folglich die Laien den gestaltenden Expert*innen ein gewisses Vertrauen entgegenbringen müssen.

Dies führt vor dem Hintergrund der im Teil I ausgeführten spezifischen Besonderheiten zu einer kritischen Rolle von Vertrauen und birgt schließlich eine Kritik der Macht der Expert*innen. Diese findet Ausdruck in der Legitimationskritik der Free Software Foundation und mündet in einer institutionalisierten Öffnung des Softwarecodes als Repräsentation der in Software eingeschriebenen Regeln.

Die prinzipielle Aufhebung der Differenz zwischen Hersteller-Expert*innen und Laien-Nutzer*innen, also die Delegitimierung der Autorität einer exklusiven Entwickler-Elite, führt zur Frage, wie unterschieden werden kann, wer sinnvoll zum gemeinsamen Wissensprodukt beitragen kann und wer nicht. Dieses „Extensionsproblem“ (Collins/Evans 2002), also die Frage der Selektion von Beitragenden und Beiträgen in Anbetracht eines erweiterten Kreises potenziell Beitragender, lösen Freie-Software-Gemeinschaften in den jeweiligen Projekt-Communities durch spezifische Mechanismen der Regulierung der Differenz zwischen Expert*innen und Laien bzw. der Konstruktion von Usern und Entwickler*innen. Bevor ich diese als Konstrukte *epistemischer Regime* betrachte, benötige ich einen differenzierteren Begriff von Expertise. Dafür mache ich Collins' und Evans' (2007) Vorschlag eines gestuften Expertise-Begriffs fruchtbar. Somit lassen sich die verschiedenen Grade von Expertise, wie sie für deren Produktionsmodus in Freie Software Communities konstitutiv sind, fassen und operationalisieren.

5 Differenzierte Expertisegrade und die Erweiterung der Partizipation

Entgegen der im vorigen Kapitel diskutierten Dichotomie von Expert*innen und Laien entwerfen Collins und Evans (2007) eine differenzierte Abstufung von Expertenwissen. Dies soll helfen, um die Delegation der Autorität der Expert*innen infolge der *Legitimationskrise* (Abschnitt 1.4) zu überwinden und eine Grundlage zu schaffen, um für die Diskussion über politische Entscheidungen die richtigen Akteure zu selektieren. Infolge einiger Besonderheiten von Software (vgl. Kap. 3), ergeben sich Gründe, die Legitimität der sonst für Technik charakteristischen Dichotomie von Hersteller-Expert*innen und Laien-Nutzer*innen infrage zu stellen.

Software ist nicht nur eine Technik, sondern auch eine Form von Wissen, weshalb hier auch Konzepte aus der Wissenschaftsforschung anwendbar sind. In ihrer inneren Logik sind Algorithmen formale Beschreibungen von Lösungswegen. „Ein Algorithmus ist ein Verfahren, das in einer endlichen Anzahl von elementaren Operationsschritten, deren Abfolge im Voraus in einer endlich langen Beschreibung eindeutig festgelegt ist, die Lösung eines (mathematischen) Problems erlaubt“ (Heintz 1993: 72). Software ist eine performante Form von Wissen, die sich unmittelbar als Technik anwenden lässt und somit ihre Wirkung entfaltet. Software ist also Wissen *und* besitzt eine technische Wirksamkeit, indem sie als Technikprodukt in Erscheinung tritt. Nicht zuletzt ist Software als symbolbasierte Technik vielfältigbar (im Sinne einer nicht vorhandenen Konkurrenz im Konsum beziehungsweise einer non-materiellen Form) und ohne die Notwendigkeit materieller Bauteile und mechanischer Einwirkung modifizierbar.

Im Folgenden werde ich das Kernelement der Formen von *Specialist Expertise* nach Collins und Evans (2007) herausgreifen und anschließend in Bezug setzen zur Rolle von Expert*innen in der Technikgestaltung. Dazu werde ich diese Formen von Expertise nutzbar machen, um die Varianz an Expertisegraden, die sich in FLOSS-Communities aus der Zugänglichkeit des Quelltextes ergeben, zu erfassen.

5.1 Abstufungen einer „Specialist Expertise“

Nach Ansicht der Autoren ist es nach einer langen Phase der Problematisierung wissenschaftlichen Wissens durch die *Sociology of Scientific Knowledge* (vgl. Abschnitt 1.4) schließlich an der Zeit, ihre Erkenntnisse konstruktiv zu wenden, um legitime und valide Expertise für anstehende Entscheidungen zu klassifizieren und in zeitlich kritischen Entscheidungen auch vor Eintreten eines wissenschaftlichen Konsenses die *richtigen Argumente* zu berücksichtigen (vgl. Collins/Evans 2002: 241).

Dieser Anspruch begründet für sie den Ausgangspunkt für eine konstruktive *Sociology of Expertise and Experience*. Durch sie soll das Erweiterungsproblem („Problem of Extension“) – die Frage wer sinnvollerweise in Entscheidungsprozesse integriert werden soll, wenn nicht allein renommierte Expert*innen – gelöst und das gesellschaftliche Vertrauen in Expertise wieder hergestellt werden.

Im Zuge ihrer Begründung einer „normativen Theorie der Expertise“ unterscheiden Collins und Evans (2007: 18ff.) im Wesentlichen fünf Formen von *Specialist Expertise*. Diese sollen dazu dienen, um in der Erweiterung des Kreises von an wichtigen Entscheidungen beteiligten Menschen Kriterien zu beschreiben, um diejenigen zu selektieren, die im Besitz des notwendigen Fachwissens (Expertise) oder wertvollen Erfahrungswissens (Stakeholder) sind:

- *Contributory Expertise* bezeichnet Expertise, die dazu befähigt, zum aktuellen Wissensstand in einem Gebiet beizutragen
- *Interactional Expertise* bezeichnet Expertise, die notwendig ist, um sich auf Augenhöhe mit den anderen am Forschungsgegenstand arbeitenden auszutauschen, und impliziert die Beherrschung der spezifischen Fachsprache
- *Primary Source Knowledge* ist Wissen, das aus der Rezeption direkter Wissensquellen stammt, also etwa wissenschaftlicher Artikel
- *Popular Understanding* ergibt sich aus der Rezeption von Massenmedien und Populärliteratur zu einem spezifischen Gegenstand (in Abgrenzung von wissenschaftlichen Publikationen)
- *Beer-Mat Knowledge* bedeutet schematisches Wissen über die Materie, das ähnlich einer Kurz-Erklärung in einem Wissensspiel auf einem

„Bierdeckel“ (oder bspw. auf der Pappe einer Zigarettenpapierschachtel) erklärt werden kann.

Ungeachtet der Frage nach einer ontologischen Validität dieser Kategorien stellt diese Kategorisierung ein hilfreiches Gerüst für die Analyse der Partizipation in der Rezeption und Produktion von Wissen dar, wie ich im Folgenden zeigen werde.

Diese Kategorien lassen sich für die Betrachtung der FLOSS-Produktion sinnvoll nutzbar machen. Hier gibt es keine den Projekten übergeordnete Autorität, die Expertise legitimiert, sondern Beiträge zur gemeinsamen Wissensbasis werden bezüglich ihrer Nützlichkeit bewertet und die Zuschreibung von Expertise wird jeweils innerhalb einer Community ausgehandelt. Die Möglichkeit des Beitrags hängt aber durchaus ab von der vorhandenen Expertise, insofern, dass Beiträge, die nicht anschlussfähig erscheinen, abgewiesen werden. Dies aber hängt im Detail ab von der spezifischen Konfiguration des in der jeweiligen Community herrschenden epistemischen Regimes, wie ich weiter unten ausführen werde (s. Kap. 6). Diese epistemische Konfiguration der Communities implizieren wiederum Autoritäten, deren Legitimation und Formalisierung aber sehr unterschiedlich sein können.

Wie in Kapitel 1 ausgeführt, ergab sich die Trennung zwischen Software-Hersteller*innen durch Lizenzregime, die im Zuge der Genese eines Softwaremarktes entstanden sind und die darauf ausgerichtet waren, den Quelltext einer Software unter Verschluss zu halten, um das darin vergegenständlichte Wissen den anderen Marktteilnehmern als Wettbewerbsvorteil vorzuhalten. Anders als in der Wissenschaft ergab sich also die Schließung der wissenschaftlichen Wissensproduktion im Software-Bereich durch die Entwicklung eines Software-Marktes und die damit einhergehende Proprietarisierung der Software. Im Unterschied zur Wissenschaft wurde die exklusive Softwareproduktion, also die Entwicklung in einem erlesenen Kreis von Programmierer*innen, von Beginn der Etablierung der geschlossenen Softwareproduktion an kritisiert, und zwar durch Beitragende (in diesem Fall selbst Wissenschaftler*innen, die Computer für ihre Zwecke nutzen), die durch die Einführung einer proprietären Lizenz an der Mitarbeit am Wissensprodukt – an dem sie vormals selbst mitgewirkt hatten –, nämlich den Quelltext des Betriebssystems UNIX, ausgeschlossen wurden (s. S. 29).

Diese Entwicklung von offener Innovationsphase zur kommerziellen Schließung im Zuge der Produktwerdung ist und war keineswegs außergewöhnlich. Auch in den frühen Tagen der Computer-Innovation wurden etwa

im Homebrew-Computer Club zunächst offen Innovationsideen diskutiert, die dann später in ein kommerzielles Produkt überführt wurden. Folglich blieben die Details der Konstruktion und spezifischen Konfiguration als exklusiv betrachtetes Wissen unter Verschluss, um einen Wettbewerbsvorteil zu bewahren (vgl. Schrape 2016). Jedoch regte sich in der Freien Software Community ein gewisser Widerstand gegen die kommerzielle Schließung, was dann in der Folge zur Gründung der Free Software Foundation führte (ausführlicher dazu s. Kap. 1).

Aufgrund der freien Lizenzen, die per Definition jedem und jeder die Möglichkeit der Veränderung der Software einräumen, ist also zunächst eine Beschränkung der Änderungen durch Autoritäten nicht vorgesehen. Praktisch ergibt sich jedoch durch die Organisationsform einer jeden Community doch eine gewisse Schließung, da für die gemeinsame Code-Basis explizite Zugriffsrechte zugeteilt werden, dies aber nur für ausgewählte Personengruppen (genauer gesagt sind die Schreibrechte beschränkt, denn Leserechte stehen ebenfalls per Definition allen zu). Ungeachtet dessen ist es zwar allen gestattet, für sich (lokal auf ihrem Computer) ihre eigene Software zu basteln, aber bei jeder Modifikation besteht generell ein Interesse der Autor*innen, die Änderung in die gemeinsame Code-Basis einfließen zu lassen – insbesondere, weil ansonsten, nach einem Update aus dem gemeinsamen Pool, die spezifischen individuellen Änderung wieder von neuem in den Code eingearbeitet werden müssten.

Für die Zuteilung von Schreibrechten messen FLOSS-Communities die Expertise ihrer Mitglieder nicht nach der Anerkennung durch eine zentrale Autorität, sondern an den erbrachten „Leistungen“ für die Community (Meritokratie). Die Anerkennung der Leistung beruht dabei maßgeblich auf einer Bewertung der Beiträge und der damit verbundenen Expertise. Für die Analyse der unterschiedlichen Ausprägungen und Bedeutungen verschiedener Grade von Expertise werde ich die von Collins und Evans beschriebenen Expertise-Typen nutzbar machen. Mit diesen Begriffen können die Differenzen der betrachteten Regime deutlich herausgearbeitet werden.

Collins und Evans (2007) unterscheiden dabei begrifflich zwischen *Expertise* und *Knowledge*. Hier kann man anknüpfen an die im vorigen Kapitel beschriebene Unterscheidung von Expert*innen, die die Funktionsweise der Technik durchdringen, und Laien, die diese nicht im Detail begreifen, sondern eher eine Form von *Benutzungswissen* besitzen. Der Begriff des Laien ist hierbei etwas irreführend, handelt es sich in unserer Betrachtung doch sehr wohl um Mitglieder der Community mit teils tiefgreifendem Wissen. Aller-

dings verstehe ich in diesem Zusammenhang unter Expertise die Fähigkeit, Softwarecode zu lesen und zu verstehen – also die Fähigkeit, zum Code beizutragen (*Contributory Expertise*) –, oder zumindest die Fähigkeit, mit Entwickler*innen, die diese technische Expertise haben, auf Augenhöhe zu kommunizieren (*Interactional Expertise*). Die Formen von „Knowledge“ entsprechen eher einem Wissen auf Anwendungsebene (*Primary Source Knowledge* bis *Beer-Mat Knowledge*).

Dadurch gewinnen wir in der Übertragung der Begriffe Abstufungen unterschiedlicher Expertisegrade im Bereich des technischen Verständnisses einerseits und unterschiedliche Grade von Benutzungswissen im Bereich der Konfiguration und Anwendung andererseits, was für die Analyse der differenzierten Wissensformen in FLOSS-Communities nützlich ist:

- **Contributory Expertise** In Anbetracht der zentralen Rolle des Softwarecodes im Bereich der FLOSS verstehe ich unter *Contributory Expertise* die Expertise derjenigen Akteure, die zum Softwarecode beitragen können – Beiträge zur Dokumentation, dem Wissenspool der Community, zählen hier also nicht als qualifizierend.
- **Interactional Expertise** stellt dementsprechend die Expertise dar, die Sprache der Entwickler*innen zu beherrschen und die Funktionsweise von Software und das Zusammenspiel der Software-Komponenten (in Betriebssystemen) nachvollziehen zu können. Diese Form der Expertise spielt eine zentrale Rolle in der Welt der FLOSS, da *Interactional Expertise* hilfreich ist, gerade um Rückmeldung über Fehler zu geben und die Funktionsweise genau zu dokumentieren.
- **Primary Source Knowledge** schließlich bezieht sich auf das Wissen aus der Dokumentation der einschlägigen Nachschlagewerke. Als primäre Quelle schlechthin stellen sich die sogenannten „manpages“ dar, die fester Bestandteil jedes Linux-Programmes und im Wesentlichen über die Distributionen hinweg standardisiert sind. Sie werden üblicherweise aus der Kommandozeile heraus betrachtet. Sie referenzieren ihrerseits andere relevante verwandte Linux-Befehle und bilden somit einen Ausgangspunkt für die inkrementelle Erlangung von *Interactional Expertise*. Wikis bilden weitere einschlägige Quellen, sind aber stärker von der spezifischen Community abhängig.
- **Popular Understanding** Analog zur Populärliteratur gibt es Computerzeitschriften und entsprechende Webseiten, die anwendungsorientiert Wissen für ein breiteres Publikum aufbereiten und Anleitungen und Er-

klärungen enthalten. Dabei werden auch grundlegende Konzepte der Funktionsweise von Computern verhandelt und somit Usern vermittelt, die nicht unbedingt ein tieferes Verständnis der Funktionsweise der Software haben.

- **Beer-Mat Knowledge** Zum niederschwelligeren, schematischen Wissen zähle ich grundlegende Kenntnisse der Computernutzung. Dies fängt im Bereich der Linux-Distributionen an beim Wissen, dass es ein Betriebssystem gibt und dass und wie man dieses über eine entsprechende Installations-CD wechseln kann. Ein tieferes Verständnis von Computern ist dafür, wie ich zeigen werde, hilfreich, aber nicht zwingend nötig.

Die Anwendung der Expertise-Kategorien auf die Domain der Freie/Open-Source-Software-Welt zeigt die Ausdifferenzierung verschiedener Stufen von Expertenwissen und Wissen (oder Benutzungswissen), die in einer Community vorhanden sind, die aber je Community eine unterschiedliche Rolle spielen.

5.2 Das Selektionsproblem erweiterter Partizipation

Im Zuge der Freie-Software-Lizenzen wird der Kreis der an der Konstruktion der Technik Beteiligten geöffnet. Daraus ergibt sich als direkte Konsequenz die Frage nach einer *Grenzziehung und Selektion* der Beitragenden beziehungsweise der Beiträge, um ein lauffähiges Produkt zu erhalten, was ich im Folgenden näher ausführen werde.

Zwar erlauben die freien Lizenzen jedem User, die Software in ihrer Funktionsweise zu verändern, und gewähren so ein Recht zur totalen individuellen Modifikation und Distribution des Programms, jedoch besteht ein starkes Interesse der Beteiligten daran, ihre Veränderungen nicht nur als individuelles Angebot bereitzustellen, sondern in die gemeinsame Wissensbasis zurückfließen zu lassen.

In der Regel werden Modifikationen an Software als Patch bereitgestellt. Üblicherweise handelt es sich dabei um eine Datei, die die Information enthält, welche neuen Zeilen im Vergleich zu einer Ausgangsversion hinzuge-

fügt und welche entfernt wurden.²² Die Form des Patch stellt also die inkrementelle Veränderung des Codes dar, die sich auf alle Kopien eines Quellcodes jeweils recht einfach anwenden lässt. Fließt jedoch dieser Patch nicht in die gemeinsame Code-Basis, wird er also nicht auf die zentral für alle bereit gestellte Version angewendet, muss jeder User, der die Modifikation verwenden will, nach Bezug einer aktualisierten Version aus dem gemeinsamen Pool, vormals angewendete Patches wieder neu einspielen.²³

Für die Pflege einer *gemeinsamen* Code-Basis ergibt sich dadurch folgendes „Extensionsproblem“: Es wird notwendig, von allen Beiträgen aller User eine Selektion zu treffen, um ein funktionsfähiges Produkt zu erhalten und somit eine gewisse Qualität sicherzustellen.

Unabhängig von der prinzipiellen und grundsätzlichen Möglichkeit, den Softwarecode nach eigenen Wünschen zu verändern, die in den freien Lizenzen festgeschrieben ist, gibt es also einen Kreis von (privilegierten) Entwickler*innen, die die Code-Basis hüten und bewahren – jedoch von allerlei anderen Beitragenden Änderungsvorschläge bekommen, die sie nach eigenem Ermessen annehmen können oder nicht. Vor dem Hintergrund des Interesses der Beitragenden, ihre Modifikation in die Code-Basis integriert zu sehen, treten die Beitragenden – trotz der erbrachten Vorleistung an Energie und Arbeit – gewissermaßen als Bittsteller in Erscheinung. In Coleman (2013: 129) findet sich dazu folgendes Zitat eines BSD-Entwicklers:

There was a process by which you wrote some code and submitted in the “I-am-not-worthy but I-hope-that-this-will-be-of-use-to-you supplication mode” to Berkeley, and if they kinda looked at it and thought, “Oh, this is cool”, then it would make it in, and if they said, “Interesting idea, but there is a better way to do that”, they might write a different implementation of it.

Die Aussage illustriert, wie sich – ohne die individuellen Rechte der Nutzer*innen zu beschneiden – eine Elite herausbildet, die nach eigenen (von

22 In anderen Kontexten können Patches Binärdateien sein oder einfach Update-Pakete bedeuten. Eine sehr allgemeine Definition findet sich auf Wikipedia: “A patch is a piece of software designed to update a computer program or its supporting data, to fix or improve it” ([https://en.wikipedia.org/wiki/Patch_\(computing\)](https://en.wikipedia.org/wiki/Patch_(computing))), letzter Aufruf, 12.1.2017).

23 Dazu ist es außerdem nötig, die unkompilierten Software-Quellen zu beziehen, auf die sich der Patch anwenden lässt, um anschließend den modifizierten Quelltext zum ausführbaren Programm zu kompilieren, hingegen ist es sonst auch bei Freier/Open Source Software üblich, Software-Programme kompiliert in Binärdateien, also als ausführbare Dateien, zu beziehen.

außen nicht immer nachvollziehbaren) Kriterien entscheidet, welcher Code, welche Veränderungen an der Software angenommen werden und welche nicht. Die Kriterien, nach denen derartige Vorschläge angenommen oder abgelehnt werden, sind je nach Verfasstheit der Community mehr oder weniger explizit oder verhandelbar. Im Falle von weniger expliziten Regeln erscheint solch ein Verfahren schnell als willkürlich, was nicht gerade ein positives Klima für Beiträge begünstigt, sondern bei Entwickler*innen bisweilen Unwillen hervorruft.

1993 begründete Ian Murdock zusammen mit anderen Entwickler*innen ein Paketmanagement-System, das es ermöglichte, die individuellen Beiträge einer großen Anzahl von Entwickler*innen zu integrieren. Dies war ein wichtiger Schritt für einen transparenten Entwicklungsmodus, und eine Innovation bezüglich einer stärker formalisierten und transparenten Struktur, auf deren Grundlage die Entwickler*innen gleichberechtigt zusammenarbeiten können (vgl. ebd.: 129 ff.).

Im Laufe der Zeit verfasste das Debian-Projekt bestimmte Dokumente, die das Projekt auf gewisse Strukturen und Werte festlegten. Insbesondere wurde ein Gesellschaftsvertrag entwickelt, der Freie Software als Basis des Projekts festschreibt, sowie Richtlinien mit Kriterien zur Beurteilung, ob eine Software als Freie Software einzustufen ist, die *Debian Free Software Guidelines*, kurz DFSG. Das Projekt entwickelte gar eine Verfassung, die zentral Entscheidungsstrukturen festlegt, darunter auch die Möglichkeit, politische Entscheidungen über eine General Resolution durch alle Mitglieder des formalen Entwickler-Kreises demokratisch zu entscheiden.

Es zeigen sich also bei Debian starke Setzungen bezüglich der normativen Orientierung und ganz konkret in der Organisationsstruktur, um transparente und damit nachvollziehbare und verlässliche Verfahren zu definieren und eine bürokratische Meritokratie zu errichten. Somit setzt das Debian-Projekt der sonst teils üblichen elitären Willkür-Herrschaft eines kleinen Entwicklerkreises ein verfasstes demokratisches System gleichberechtigter Vieler entgegen. Hier zeigt sich also ein organisiertes Vorgehen zur Überwindung des Extensionsproblems. Unterstützt durch sozio-technische Systeme wie Paket-Management (Rechte-Verwaltung), Bug-Tracking-System (Aufgabenkoordination) und Kryptografie (Vertrauens-Verhältnisse) wird ein Selektionsregime der Beiträge ausgebaut. Die so geschaffene transparente Bürokratie stiftet Vertrauen seitens der Beitragenden in die Entwickler-Community darin, dass ihre Beiträge nach meritokratischen statt willkürlichen Gesichtspunkten betrachtet werden. Dabei liegt der normativen Orientierung einer

Community nicht zuletzt eine Konstruktion von Expertise zugrunde, die sich grob mit der Akzeptanz gewisser Expertise-Stufen nach Collins und Evans (2007) kategorisieren lässt, wie ich im weiteren Verlauf der Arbeit herausarbeiten werde.

Auch von Seiten der Nutzer*innen stiftet die Struktur der Community Vertrauen. Diese können aufgrund der Beschaffenheit und Sichtbarkeit der strukturellen und normativen Setzungen der Organisation einsehen, welche Werte der Software-Entwicklung zugrundeliegen – Werte, die, wie ich zeigen werde, auch im Softwareprodukt sichtbar werden. Bevor ich nun im Detail in der Empirie verschiedene Ausprägungen solcher Selektionsregime beschreiben werde, benötige ich ein theoretisches Konstrukt zur analytischen Betrachtung derselben. Dazu werde ich im nächsten Kapitel den Begriff des epistemischen Regimes aufgreifen und im Hinblick auf das empirische Phänomen weiterentwickeln.

6 Die Regulierung der Wissensproduktion durch epistemische Regime

Wie ich im vorausgehenden Kapitel argumentiert habe, gründet die Öffnung des Quelltextes von Software – der absolute Bezugspunkt aller Befürworter*innen von Freier und Open Source Software – auf einer Legitimationskrise, die das Vertrauen in eine aus dem User-Kontext *entbettete* Softwareproduktion problematisiert und damit die grundsätzliche Wissensdifferenz zwischen Expert*innen und Laien als konstitutive Dichotomie für die Softwaretechnik infrage stellt.

Aus der Auflösung der Dichotomie zwischen Expert*innen und Laien als Hersteller*innen und User ergibt sich ein Extensionsproblem im Sinne einer notwendigen Eingrenzung des Kreises der Beteiligten (infolge der Erweiterung). Für die Anfertigung eines funktionierenden Software-Programms erscheint eine gewisse Schließung mitunter notwendig, um eine performative Nutzung des in Software vergegenständlichten Wissens zu ermöglichen – und um einen gesicherten Ereigniszusammenhang zwischen Bedienung der Technik und erwartetem Ergebnis zu erhalten.

Wie ich im weiteren Verlauf der Arbeit darlegen werde, bilden sich in den betrachteten Communities formale Strukturen zur nachvollziehbaren und expliziten Regulierung der Wissensproduktion heraus, nicht zuletzt um die – wiederum zur Wissenschaft analogen – meritokratische Ideologie der Freie-Software-Bewegung zu institutionalisieren. In die sozialen Strukturen der Wissensproduktion sind dabei bestimmte Werte eingeschrieben, die spezifisch je Community sind und die diese Werte reproduzieren. Diesen Werten inhärent sind Kriterien der Zuschreibung von Kompetenz und der Legitimation von Wissen und Beiträgen.

In den folgenden Abschnitten werde ich nun verschiedene Konzeptionen des Begriffs *epistemische Regime* diskutieren und ein analytisches Modell für meine empirische Untersuchung entwickeln. Aufgrund fehlender Berücksichtigung der konkreten Praktiken in den diskutierten Modellen werde ich zurückgreifen auf Jochen Gläfers (2006) Betrachtung der gemeinschaftlichen Produktion von Wissen in der Wissenschaft. Die entwickelten Kategorien zur Analyse epistemischer Regime werden die strukturierte Untersuchung und eine vergleichende Betrachtung der gefundenen epistemischen Regime ermöglichen.

6.1 Epistemische Regime: Vorarbeiten und Konzeption des Begriffs

FLOSS-Communities sind *offene Produktionsgemeinschaften*, die Wissen generieren. Sie folgen in ihren Praktiken der Wissensproduktion epistemischen Kriterien und bilden soziale Ordnungen aus. Während unterschiedliche Communities eine Reihe epistemischer Kriterien sowie epistemische Praktiken teilen, liegen den Kriterien auch spezifische Wertsetzungen zugrunde, die zwischen den Communities differieren. *Diese Wertsetzungen spiegeln sich in den spezifischen sozialen Ordnungen wider und prägen über die Organisation der Wissensproduktion auch das produzierte Wissen.*

Um die Wirkungsweise und das Zusammenspiel von Praktiken, Strukturen und Normen zu betrachten, verwende ich den Begriff *epistemische Regime*. Die Regime steuern die Einflussverteilung (Macht) der beteiligten Akteure anhand epistemischer Kriterien und stiften durch die gemeinsame Wertstruktur Vertrauen in die Richtigkeit der Grundsätze des gemeinsamen Projekts.

Der Begriff der epistemischen Regime wird insbesondere in der Wissenssoziologie und der Wissenschaftsforschung diskutiert. Schützeichel (2012: 23) setzt den Begriff des epistemischen Regimes in Beziehung zur Forschung über Wissenskulturen und Wissensordnungen. Dabei steht diese konstruktivistische Betrachtung der Genese von Wissen in der Tradition von Ludwig Flecks Idee von verschiedenen Denkstilen, die in Denkkollektiven festlegen, „was nicht anders gedacht werden kann“ (Fleck 1979: 99), und dadurch das produzierte Wissen einer Gemeinschaft strukturieren, sowie Kuhns (1970) Beschreibung der Abhängigkeit von Wissenschaft von vorherrschenden Paradigmen. Eine Grundannahme ist dabei, dass Wissen eben nicht objektiv richtig oder falsch ist, sondern dass die Gültigkeit von Wissen immer abhängt von der sozialen Konstruktion der Wirklichkeit bestimmter Gesellschaften oder Gemeinschaften. Ausschlaggebend für die Wissenskultur eines Kollektivs sind daher soziale Strukturen. Expertise und Kompetenz sind auf der Wissensordnung basierende Zuschreibungen und beruhen daher ebenfalls auf sozialen Praktiken (vgl. Schützeichel 2010: 180).

Ein verwandter Begriff, der große Aufmerksamkeit erfahren hat, ist außerdem der Begriff der Wissenskulturen oder epistemischen Kulturen (vgl. Knorr-Cetina 1999). Bezeichnet „Wissenskultur“ doch im Grunde das Verhältnis von Wissen und Kultur, hat der Begriff noch einen weiten Bedeu-

tungshorizont erreicht (ausführlich dazu Zittel 2014). In Knorr-Cetinas „Epistemic Cultures“ geht es konkret um den Vergleich verschiedener wissenschaftlicher Disziplinen. Betrachtet Knorr-Cetina dabei gerade die praktisch kulturelle Verfassung einzelner epistemischer Kulturen, also die Gerechtigkeit oder das „So-sein“ der jeweiligen Kulturen, so spielt doch das Zustandekommen der jeweiligen Konfigurationen von Praktiken und Normen – die Regulierungsfunktion – in ihrer Betrachtung eine nachrangige Rolle.

Anne Marcovich und Terry Shinn betrachten verschiedene Regime der Wissensproduktion vor dem Hintergrund ihrer historischen Kontexte.²⁴ Dabei geht es um die Beschreibung gesellschaftlicher Tendenzen in der Bewertung und Konstruktion von Wissen und den Konsequenzen für die Wissensproduktion. Ebenso wird der Begriff des *Knowledge Regimes* (vgl. Pestre 2003; Rammert 2004) eher für die übergreifenden Wissensordnungen in Gesellschaften verwendet. Wehling (2007: 706 f.) versteht unter *Wissensordnungen* übergreifende kulturell und institutionell verankerte Rahmen und grenzt demgegenüber den Begriff der Wissensregime ab, die er auf konkrete spezifische Handlungsfelder bezieht, und räumt ihnen einen unmittelbaren inhaltlichen Eingriff in die Wissensproduktion ein.

Zusammenfassend möchte ich mich Bösch (2016) anschließen, der in den beschriebenen Konzepten die Beschäftigung mit dem „Zusammenhang von sozialen Prozessen und Wissen“ sieht und den Begriff wie folgt pointiert:

Fokussiert das Konzept der Wissensordnungen letztlich auf Ordnungen als Ergebnis von Prozessen, gewinnt das Konzept der Wissenspolitik seine Stärke durch einen gezielten Blick auf die Verknüpfung von Wissen und politischen Prozessen, und gelingt es schließlich dem Konzept der Wissensregime, diese beiden Perspektiven zu verbinden und zudem feldunabhängig auf Regulierungsformen von, mit und durch Wissen einzugehen. (ebd.: 29)

Entgegen seinem ausgefeilten Modell hybrider Wissensregime, das er für die Untersuchung risikopolitischer und innovationspolitischer Felder entwickelt, geht es mir in meinem Forschungsansatz um die spezifischen Felder einzelner Software Communities und dabei um den von Wehling (2007: 704) als Wissensregime bezeichneten „Zusammenhang von Praktiken, Regeln, Prinzipien und Normen des Umgangs mit Wissen und unterschiedlichen Wis-

²⁴ In ihrem Papier *Regimes of Science Production and Diffusion* arbeiten sie maßgeblich vier Wissensregime heraus: disciplinary regime, utilitarian regime of science and technology production and confusion, transitory science and technology regime, research-technology regime (vgl. Marcovich/Shinn 2012).

sensformen“ (Schützeichel 2010: 180), der die epistemische Ordnung in sozialen Systemen reguliert. Schützeichel zieht es vor, von epistemischen Ordnungen und Regimen zu sprechen, um „die Pluralität der involvierten Wissensformen zu betonen“ (ebd.). Das Konzept nutze ich dafür, um in den betrachteten FLOSS-Communitites die sozialen Ordnung und deren Zusammenhang mit den epistemischen Praktiken und den darunter liegenden Normen zu analysieren.

Diese Definition enthält einige wichtige Setzungen für das hier zugrunde gelegte Verständnis des Konzepts epistemischer Regime. Einmal geht es um Prinzipien und Normen im Umgang mit Wissen, also einen *normativen Rahmen*, der konstitutiv ist für die Regulierung des Wissens. Zum anderen geht es um einen *strukturierten Zusammenhang* von Praktiken und Regeln, denen die Prinzipien und Normen inhärent sind. Des Weiteren wird durch das epistemische Regime eine Ordnung gebildet, die in der jeweiligen sozialen Form regulierend wirkt.

Schützeichel (ebd.) präzisiert Wehlings Definition und verortet die Regulierungsfunktion epistemischer Regime auf drei Ebenen. Demnach legen sie erstens fest, *welche Akteure legitimiert sind*, Wissen und Kompetenz zu generieren und auf diese zurückzugreifen, also das Wissen (und hier auch: die Software) zu konsumieren (Sozialdimension). Zweitens definieren sie, *welche Problembereiche verhandelt werden*, also welche Beiträge in der Diskussion, Kommunikation (und hier: Programmierung) als valide gelten (Sachdimension). Drittens *legen sie Kriterien fest*, die für die Selektion von Wissen und Kompetenz gelten (Temporaldimension).

Es geht hier in dieser Definition also einerseits um die Klassifikation der an der Wissensproduktion (und -konsumtion) beteiligten *sozialen Akteure* und andererseits um die Definition des *gemeinsamen Arbeitsgegenstandes* und damit um eine Abgrenzung dessen, was nicht zur gemeinsamen Bearbeitung gehört. Schließlich liegt dieser Rahmung ein normatives Gerüst zugrunde, das aus *epistemischen Kriterien* besteht, die festgelegt sind und die Wissen und Kompetenz als solches ausweisen und legitimieren und somit ein geteiltes Verständnis zugrunde legen, was als Wissen und Kompetenz anerkannt und akzeptiert wird – und was nicht. Die epistemischen Kriterien sind dabei Gegenstand der gemeinschaftlichen Aushandlung und damit ist auch das Verständnis dessen, was als Wissen und Kompetenz gilt, verhandelbar.

Schützeichel (ebd.: 181) stellt fest, dass eine wichtige Funktion epistemischer Regime ist, „dass sie mit Hilfe öffentlicher Kriterien gewisse Standards der Zuschreibung für ‚Wissen‘ oder ‚Kompetenz‘ definieren“. Wie transpa-

rent die Kriterien im Einzelfall sind, mag dahingestellt sein, aber die grundsätzliche Sichtbarkeit und Verhandelbarkeit dieser Kriterien spielt in meritokratischen Gemeinschaften wie den hier betrachteten eine wichtige Rolle, nicht zuletzt auch in Bezug auf das Vertrauen in die Kompetenz der Expert*innen. Diese Vertrauen stiftende Funktion ist virulent im Hinblick auf das oben beschriebene Legitimationsproblem. Hinzu kommt in den betrachteten Fällen die Wahlfreiheit zwischen verschiedenen epistemischen Regimen, die den Nutzer*innen überlassen, welcher Community sie ihr Vertrauen schenken möchten. In den bisher genannten Ansätzen findet jedoch die Frage danach, wie die Gültigkeit von Wissen und Kompetenz über die *Praktiken* innerhalb der Strukturen der Gemeinschaft ausgehandelt wird, keine Berücksichtigung. Für eine Untersuchung der epistemischen Regime in der Wissensproduktion ist aber die Berücksichtigung eben dieser Praktiken notwendig.

Folglich verstehe ich unter epistemischen Regimen Regulierungs-Geflechte von üblichen Praktiken der Wissensproduktion und impliziten und expliziten Verfahren und Prinzipien (soziale Strukturen), die durch die Selektion von Beiträgen und Mitgliedern die Kriterien für Wissen und Kompetenz innerhalb der Wissen produzierenden Gemeinschaft normativ festlegen.

Mit den Praktiken der Wissensproduktion setzt sich Gläser (2006) in seiner empirischen Betrachtung der sozialen Ordnung *wissenschaftlicher Produktionsgemeinschaften* auseinander. Diese möchte ich im folgenden Abschnitt nachvollziehen, um sie mit Rückgriff auf die bisher referierten Konzeptionen von epistemischen Regimen für meine empirische Untersuchung fruchtbar zu machen und im Anschluss Kategorien für eine analytische Betrachtung auszuarbeiten.

6.2 Die soziale Ordnung der Wissensproduktion

Jochen Gläser zielt mit seiner Arbeit zur Produktion wissenschaftlichen Wissens (2006) auf die Analyse der sozialen Ordnung der Wissensproduktion. Dabei schafft er mit Blick auf empirisches Material über verschiedene Zusammenhänge der wissenschaftlichen Wissensproduktionen einen Vergleichs- und Analyserahmen, den ich hier kurz skizzieren möchte. Als generalisierte Merkmale kollektiver Produktionssysteme identifiziert Gläser im Wesentlichen vier Elemente, die die soziale Ordnung ausmachen. Das erste

ist dabei die *Mitgliedschaft*, da sie die Zugehörigkeit zum produzierenden Kollektiv definiert. Dabei spielen Aspekte wie individuelle Motivationen und die Konstitution von Mitgliedschaft für das spezifische Kollektiv eine wichtige Rolle.

Ähnlich wie in FLOSS-Projekten definieren sich ihre Mitglieder im Wesentlichen durch die Wahrnehmung und Bearbeitung eines gemeinsamen Arbeitsgegenstandes (vgl. ebd.: 274), was zur Folge hat, dass sich flüchtige und unscharfe Grenzen der Gemeinschaft ergeben und daher nicht immer alle Mitglieder bekannt sind. Darüber hinaus besteht eine starke Indifferenz der Gemeinschaft gegenüber den Motivationen der einzelnen Entwickler*innen.²⁵ Ebenfalls gemeinsam ist den beiden Arten von Gemeinschaften, dass die eigentliche Produktion des Wissens zunächst losgekoppelt vom Verdienst für den Lebensunterhalt stattfindet, wenngleich einige die gemeinsame Arbeit im Zusammenhang von Lohnarbeit bewerkstelligen. Dies begünstigt lokale, dezentrale Entscheidungen und führt zu einer grundsätzlichen Freiwilligkeit der Beitragenden bei der Wahl ihrer Aufgaben. Das wiederum bedeutet, dass es keine Zwangsmittel zur Durchführung bestimmter Aufgaben gibt.

Die anderen drei Ordnungsmerkmale beziehen sich – im Gegensatz zur Mitgliedschaft – nun weniger auf die Akteure, sondern eher auf den Arbeitsgegenstand. Gläser betrachtet dazu die *Formulierung, Bearbeitung und Integration von Aufgaben* (vgl. ebd.: 272 ff.).

Hier beobachtet Gläser ebenfalls Parallelen zwischen Wissenschaft und FLOSS bezüglich der selbstverantwortlichen Orientierung am gemeinsamen Arbeitsgegenstand bei der Formulierung, Zuteilung und Bearbeitung von Aufgaben. Hier wie dort finden und formulieren die Mitglieder der Produktionsgemeinschaft ihre Aufgaben selbst, und zwar so, dass sie sie bearbeiten und bewältigen können. In Software-Projekten geschieht dies in Bezug auf den Softwarecode explizit vermittelt über sogenannte Bug Tracking Tools, mit denen Fehler („Bugs“) und Änderungswünsche („Feature Requests“) aufgegeben werden können und die Arbeit daran dokumentiert wird. Auch die *Integration* der Beiträge in der Softwareproduktion findet wie in der Wissenschaft über ein *Peer Review* statt. Beiträge werden erst von anderen Kol-

25 Coleman (2013) arbeitet in ihrer Studie über Freie-Software-Hacker heraus, dass dort eine Art „politischer Agnostizismus“ verbreitet ist, der genau diese Indifferenz, hier gegenüber politischen Motivationen, beinhaltet.

leg*innen überprüft und gegebenenfalls Änderungen eingefordert, bevor der Beitrag in den gemeinsamen Wissenspool fließen darf.

Gläser stellt einen markanten Unterschied zwischen Wissenschaft und Software darin fest, dass durch „Nur-Nutzer“ von Software Produktion und Anwendung auseinanderfallen (vgl. ebd.: 277), es also einen Funktionsanspruch gibt, der es Laien erlaubt, ausschließlich als Konsument auf das Wissensprodukt zuzugreifen. Hierdurch erweitert sich der Kreis der Mitglieder der Produktionsgemeinschaft, zu der im Bereich der Freien/Open Source Software auch die passiven Mitglieder zählen, da deren Passivität prinzipiell nur ein temporärer Zustand ist. Viele Nutzer engagieren sich in einem konkreten Punkt aus eigenem Interesse und tauchen früher oder später wieder in die Passivität ab, was sich aber zu einem gegebenen Zeitpunkt wieder ändern kann (vgl. auch Bagozzi/Dholakia 2006).

Die beschriebenen Ordnungsmerkmale finden sich also sowohl in Wissenschaft als auch in der offenen Softwareproduktion und stellen hilfreiche Kriterien für die Analyse der Wissensproduktion dar. Im meinem Vorhaben geht es mir aber insbesondere um die Rolle von normativen Setzungen, die in die soziale Ordnung mit einfließen – und wie sie wiederum auf das Produkt zurückwirken.

Gläser's Analyse zeigt die Bedeutung des Definitionsbereichs der beteiligten Akteure (Mitgliedschaft) sowie die Relevanz der Koordination der Produktion von Beiträgen für die Ordnung innerhalb der Gemeinschaft. Dafür ist nicht zuletzt die Rolle von Institutionen und Regelsystemen zu berücksichtigen (vgl. Gläser 2006: 72).

6.3 Dimensionen epistemischer Regime

Wehling bleibt mit seiner Beschreibung des Begriffs epistemischer Regime mit „Praktiken, Prinzipien und Normen“ zunächst sehr abstrakt. Darin werden sowohl praktische und informelle Regeln impliziert als auch explizite Verfahren und Prozesse. Zudem enthält der Verweis auf Normen den Hinweis auf die Relevanz der definitorischen Setzung von Kompetenz und Expertise innerhalb des Regimes.

Schützeichels Übertragung der Sach-, Sozial- und Temporaldimension enthält darüber hinaus den Hinweis auf die Rolle der als zugehörig erachteten

Akteure, ihre Legitimation und Autorität sowie die Bedeutung der Setzung des gemeinsamen Gegenstandes beziehungsweise der Validität von Beiträgen. Dies impliziert wiederum Kriterien, nach denen Mitglieder und Beiträge als legitim und valide selektiert werden können.

Gläser liefert einen Analyserahmen für die empirische Betrachtung der Ordnung der Wissensproduktion. Sowohl die Mitgliedschaft in der Gemeinschaft als auch die Koordination von Formulierung, Bearbeitung und Integration der Aufgaben in der Wissensproduktion spielen eine Rolle innerhalb des epistemischen Regimes.

Meine hier vorgestellte Konzeption epistemischer Regime zielt insbesondere auf die normative Prägung des Regimes im Sinne einer a priori gesetzten Definition von Kompetenz und Expertise und ihrer Wirkung auf das produzierte Wissen. Dies impliziert eine starke Wertsetzung, die sowohl in die sozialen Ordnungsstrukturen als auch in das produzierte Wissen selbst eingeschrieben wird. Das heißt, in meiner analytischen Betrachtung gehe ich aus von einem Wertekonsens, der dem epistemischen Regime zugrunde liegt; dieser wirkt strukturierend, wird aber dennoch durch das situative Handeln der einzelnen Akteure marginal beeinflusst.

Mitgliedschaft im Sinne von Gläser ist ein strukturelles Element der Ordnung. Als solches betrachte ich die *formalen Mitgliedsverfahren* der Communities. Eine andere Facette von Mitgliedschaft bildet eine Art *Selbstzuschreibung* des Individuums zum Kollektiv. Dies geht einher mit der Identifikation mit den innerhalb einer Community praktizierten und kommunizierten Werten und repräsentiert daher ein gewisses Selbstverständnis der Community und ihrer Mitglieder. Dabei fließt mit ein, wer aus der Perspektive der Community als legitimes Mitglied betrachtet wird im Sinne eines idealen Selbstbildes der Community, oder anders formuliert, eines projizierten Nutzerbildes, das sich kollektiv in der Community formt und als normativer Referenzpunkt dient.

In dieser Hinsicht verstehe ich Mitgliedschaft als eine unscharfe, selbst induzierte Zugehörigkeit zur Community. Durch die Nutzung der Software ergeben sich Möglichkeiten der Interaktion mit der Community und mit der zunehmenden Interaktion auch Möglichkeiten des Beitragens und dadurch eine aktive Community-Rolle. Zu einem guten Teil ist Mitgliedschaft folglich ein Zugehörigkeitsgefühl zur Community, das einhergeht mit einer Identifikation mit den dort zu grundlegenden Werten.

Dieses reflexive Nutzerbild spiegelt sich wider in der formalen Mitgliedschaft und den Rollen einer ausgestalteten formalen Organisation, ist aber

auch Element einer diskursiven Verhandlung innerhalb der Community und wird mit der Legitimation von Beiträgen oder eben der Nicht-Legitimation bzw. Delegitimation von Beiträgen – und damit der jeweiligen beitragenden Akteure – ko-konstituiert.

Mitgliedschaft als kollektives Selbstbild

Mitgliedschaft ist aufgrund der Selbstzuschreibung der einzelnen Akteure zunächst eine schwammige und schwer einzugrenzende Kategorie. Jedoch gibt es dabei implizite und explizite Zuschreibungen der bestehenden Mitglieder gegenüber potenziellen Mitgliedern, die auf Grundlage der geteilten Werte eine sozial selektive Wirkung haben.

Eine Besonderheit der hier betrachteten Communities – gerade in Abgrenzung von wissenschaftlichen Communities – ist, dass es durch die gemeinsamen Kommunikationswerkzeuge wie Internet Relay Chats, Mailinglisten und Foren eine direkte interaktive Ebene gibt, die bis hin zu einem synchronen Dialog reicht, aber in allen Ebenen informell und dennoch schriftlich und in der Regel (halb)öffentlich stattfindet. In der Wissenschaft beschränken sich Mailinglisten häufig auf die Zirkulation von Ankündigungen und werden eher selten als Diskussions-Plattform genutzt. Im Gegensatz dazu bekommen die hier fokussierten Communities einen direkteren Charakter, da die anderen Mitglieder durch die direkte kommunikative Interaktion permanent sichtbar und erlebbar sind und nicht etwa nur über Publikationen in größeren Abständen „in Diskurs treten“. Dies führt dazu, dass das Selbstbild der Community und das Gefühl der Zugehörigkeit eine besondere Relevanz gewinnen, die sich auch in der Rolle der Mitgliedschaft widerspiegelt. Zudem werden durch die interaktive und explizite Kommunikation Selektionsmechanismen deutlicher sichtbar und erfahrbar als in der Wissenschaft.

Jochen Gläser führt zur Betrachtung der Mitgliedschaft verschiedene Fragen an: Wovon hängt die Mitgliedschaft ab, welches soziale Phänomen konstituiert die Mitgliedschaft und wie werden Akteure dazu motiviert, teilzunehmen und aktiv beizutragen? Übersetzt auf die Betrachtung epistemischer Regime ergeben sich dabei folgende Dimensionen:

Mitgliedschaft konstituiert sich einerseits dadurch, dass interessierten Akteuren der *Einstieg* ermöglicht wird. Dies kann unterstützt werden durch Mentor*innen und verschiedene Einstiegsangebote auch niederschwelliger Art, eine Community kann sich aber auch stärker abgrenzen – beispielsweise durch eine striktere Diskussionskultur und höhere Anforderungen an die Einsteiger*innen – und somit eine stärkere Auslese betreiben. Dabei können

Einstiegsangebote (oder auch das Fehlen von dezidierten Angeboten) verschiedene Individuen adressieren und für diese unterschiedliche Rollen und damit verbundene Aufgaben vorsehen oder nahelegen – oder aber diese eher sich selbst überlassen. Die Anforderungen oder unterschiedlichen *Voraussetzungen*, die von potenziellen Mitgliedern erwartet werden, beziehen sich auf deren Wissen und Fähigkeiten, aber auch auf andere Eigenschaften wie beispielsweise Lernbereitschaft oder Commitment. Dies wiederum beeinflusst auch die Motivation der potenziellen Mitglieder durch die Erwartung von gewissen Kompetenzen und Fähigkeiten und steckt somit auch den Rahmen für die anschließende aktive Beteiligung (s. nächster Abschnitt).

Voraussetzungen und Einstiegsmöglichkeiten zeigen sich auch in der Definition der *formalen Mitgliedschaft*. Diese ist häufig ausdifferenziert in verschiedene Rollen. Hier betrachtet werden sollen vornehmlich die Rollen, an die die geringsten Anforderungen gestellt, die aber dennoch formal von der Community validiert werden, etwa durch ein formales Verfahren.

Somit werden gewisse *Nutzerbilder* nahegelegt oder angesprochen, die im betrachteten Falle von sporadischen Laien-Nutzer*innen über Administrator*innen bis hin zu Programmierer*innen reichen.

Strukturen und Verfahren der Community

Ein weiteres Augenmerk ist gerichtet auf die institutionalisierten Prozesse und Verfahren – oder allgemeiner: die *Strukturen*, die in der jeweiligen Community wirksam sind. Auch hier beleuchte ich wiederum weniger die Mechanismen der Ordnungsbildung selbst als die Wirkung ihrer ordnenden Mechanik beziehungsweise die in sie eingeschriebenen und darin wirksamen normativen Ordnungen.

Dabei betrachte ich die *Akteure, die an Entscheidungen beteiligt sind*, was je nach sozialer Ordnung formale Rollen einer Organisation sein können oder auch dezentrale Akteure, die allein durch ihre faktische Produktion Entscheidungen direkt in das gemeinsame Wissensprodukt einarbeiten. Dabei sind marginale Entscheidungen bei der Wahl von Alternativen (z.B. Implementierungsmöglichkeiten) in der Wissensproduktion auch „Entscheidungen“, da das Gesamtprodukt der Community sich aus derartigen einzelnen Setzungen konstituiert.

Diese Ad-hoc-Entscheidungen der Entwickler*innen bilden eine von verschiedenen Möglichkeiten von Verfahren der *Entscheidungsfindung*. Daneben gibt es von Fall zu Fall auch Wahlverfahren oder die Delegation von Entscheidungen an Gremien.

Die Auswirkungen bestimmter Festlegungen wirken sich aus auf *Entscheidungsräume* in Abhängigkeit von der Position oder Rolle der Entscheidungsträger*innen bzw. Beitragenden, findet teils aber auch unabhängig von Rollen statt.

Die *Legitimation der Autorität* der Beitragenden hängt ab von der formalen Rolle der Beitragenden sowie ihrer Position in der Community und bestimmt die Gewichtung ihrer Argumente. Dafür spielen ebenfalls informelle Faktoren wie Erfahrung und Anerkennung eine Rolle.

Schließlich unterscheiden sich die *Kriterien für die Entscheidungsfindung*, die innerhalb der Community argumentierbar sind und infolgedessen berücksichtigt werden.

Beiträge zum gemeinsamen Wissensprodukt

Ein zentrales Element einer offenen Produktionsgemeinschaft bildet schließlich der *Beitrag zum gemeinsamen Wissensprodukt*, der auch bei Gläser im Zentrum der Ordnungsmechanismen steht: die Formulierung, Bearbeitung und Integration der Beiträge. In der Fokussierung auf das Regime konzentriere ich mich hierbei auf die Validität und Legitimität von Beiträgen – also der Bewertung und Anerkennung der Beiträge durch *Mitglieder der Community*. Hier geht es mir also weniger um die objektive Betrachtung und Herausarbeitung der Mechanismen der Integration als um die konkrete Wirkung dieser Mechanismen auf die Reproduktion einer normativen Ordnung innerhalb der Produktionsgemeinschaft.

Tab. 6.1: Dimensionen der Variablen epistemischer Regime

Mitgliedschaft als kollektives Selbstbild	Strukturen und Verfahren	Beiträge zum gemeinsamen Wissensprodukt
Einstieg	entscheidende Akteure	Art der Beiträge
Voraussetzungen	Entscheidungsfindung	Anforderungen an die Beitragenden
Nutzerbild	Entscheidungsräume	beitragende Akteure
formale Mitgliedschaft	Legitimation und Kriterien	Zuweisung von Aufgaben

Dabei spielt eine wichtige Rolle, welche Akteure Beiträge zur Verfügung stellen, also wer die *beitragenden Akteure* tatsächlich sind und welche Position oder Rolle sie innerhalb der Community haben. Hier kann es beträchtliche Unterschiede geben, wer hier als legitimer Akteur auftritt und als sol-

cher akzeptiert wird. Darüber hinaus variieren allein die möglichen Positionen innerhalb der Community. Außerdem kann die *Art des Beitrags* deutlich variieren. Im betrachteten Feld kann dies von technischem Programmiercode über praktische Anleitungen bis hin zu kulinarischer Verpflegung auf einem Messestand reichen.

Hierbei sind unterschiedliche *Anforderungen an die Beitragenden* gestellt und die Beiträge werden unterschiedlich bewertet, je nach der normativen Verfassung des Regimes. Auch die Art, wie die Beiträge eingebracht werden können, ist dabei verbunden mit gewissen kognitiven Anforderungen der Bedienbarkeit und des Verständnisses der entsprechenden Werkzeuge. In manchen Fällen hängt dies auch davon ab, welche expliziten Berechtigungen der oder die Beitragende hat, die wiederum mit formalen Rollen verbunden sind.

Der Aspekt der Motivation der Mitglieder spiegelt sich an dieser Stelle wider in der *Zuweisung von Aufgaben*, die zwar grundsätzlich freiwillig selbst zugewiesen werden, aber je nach Organisationsform auch in definierte Zuständigkeiten (etwa die Betreuung eines spezifischen Softwarepakets) fallen oder gar im Falle einer Weisungsabhängigkeit (etwa durch Lohnverhältnisse) zugewiesen werden. Dabei gibt es auch unterschiedliche Herangehensweisen, um neuen Beitragenden niederschwellige Kontributionsangebote nahezu legen.

6.4 Operationalisierung zur Analyse epistemischer Regime der Softwareproduktion

Wie gezeigt, gibt es deutliche Parallelen zwischen FLOSS-Gemeinschaften und wissenschaftlichen Gemeinschaften in Betrachtung eines übergeordneten Rahmens *offener Produktionsgemeinschaften* (vgl. Gläser 2006).

Dies hängt damit zusammen, dass Software eben nicht nur eine Form von Technik ist, ein performantes Produkt, das man benutzen kann, sondern auch eine Form formalisierten Wissens, insofern Algorithmen formalisierte Lösungswege für konkrete Probleme darstellen. Code ist Wissen und das Wis-

sen der Anwendung des Codes ist auch Teil der gemeinschaftlichen Wissensbasis.²⁶

Die Mitglieder der Community modifizieren sowohl Code als auch die zugehörige Dokumentation, die notwendig ist, um den Code zu verstehen und nutzbar zu machen. Die Beiträge in der gemeinschaftlichen Produktion beinhalten daher beide Bereiche und Beitragsvorschläge werden schließlich von anderen diskutiert und modifiziert – eine Art Peer Review, wie dies in der Wissenschaft auch üblich ist. Dies geschieht nach impliziten und expliziten Regeln, die teils schriftlich festgelegt sind, teils Regeln der gemeinsamen Praxis darstellen. Diese Regeln sind ausgehandelte Strukturen. Sie sind zunächst gegeben, werden aber durch die Praxis restrukturiert. Auch die formalen Regeln werden interpretiert und damit in die Praxis eingebettet und gegebenenfalls auch reformuliert. Zunächst betrachte ich den Ist-Zustand der vorgefundenen Regeln, die sich aber über die Zeit ausgebildet haben oder auf vorherigen Überlegungen und Erfahrungen ausgearbeitet wurden. Dabei haben verschiedene Communities spezifische Vorstellungen über Nutzer und Nutzungen, die sich in der formalen und informellen Struktur niederschlagen. Den Zusammenhang zwischen sozialer Struktur und epistemischer Praxis betrachte ich mit dem Begriff des epistemischen Regimes.

Abbildung 6.2 zeigt meine Ausgangsannahme über das Zusammenspiel der betrachteten Variablen. Ich gehe davon aus, dass sich das Selbstbild der Community über die Mitgliedschaft definiert. Diese normative Grundierung determiniert das strukturelle Gerüst der Community-Governance und wirkt wiederum zurück auf die Konstruktion der Mitgliedschaft. Die Strukturen und Verfahren sowie die Anforderungen an die Mitglieder der Community fließen ein in die Bewertung und Anerkennung der vorgeschlagenen Beiträge zum gemeinsamen Wissensprodukt. Diese wirken sich schließlich aus auf das gemeinsame Wissensprodukt.

Das Wissensprodukt selbst, insbesondere die Software, wirkt wiederum zurück auf die Selektion der Mitglieder durch die im Produktionsprozess eingeschriebenen Werte, wie ich in Kapitel 9 zeigen werde.

26 Vgl. auch Degele (2000: 63): „Wissen *und* Technik und Wissen *als* Technik sind systematisch miteinander verknüpft: Informations- und Kommunikationstechnik steht ebenso im Kontext eines durch Software vermittelten Wissenssystems wie eines durch Wissen vermittelten Softwaresystems.“

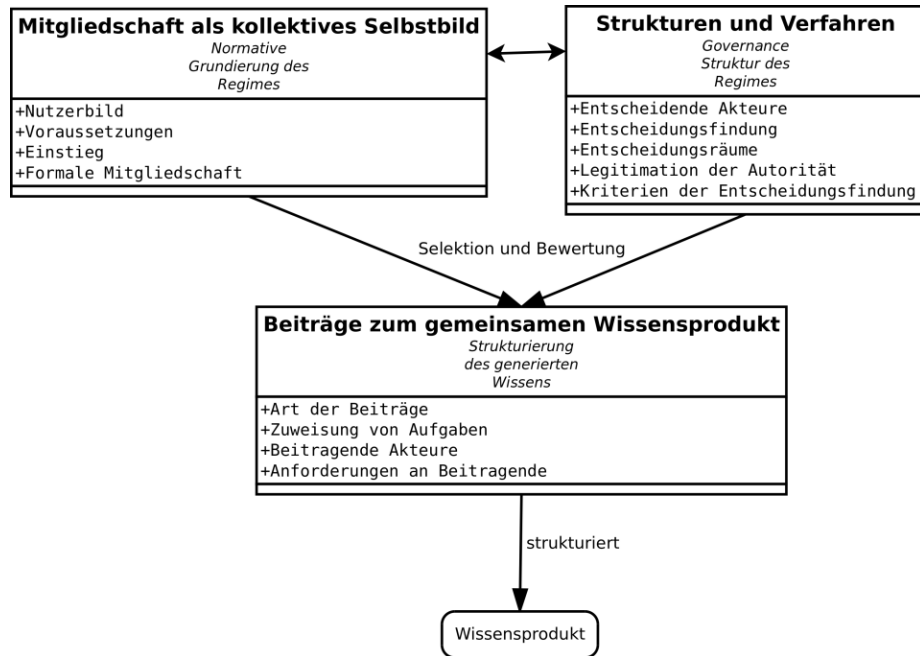


Abb. 6.2 Variablenmodell epistemischer Regime

Mit der theoretischen Ausarbeitung der Annahmen der Funktionsweise epistemischer Regime ist die Grundlage für die empirische Untersuchung gelegt, die Gegenstand des nun folgenden Teils der Arbeit ist.

Teil III

GNU/Linux Communities als epistemische Regime

7 Untersuchungsdesign

Anhand der im vorangehenden Kapitel entwickelten Variablen epistemischer Regime folgt nun eine analytische Betrachtung von verschiedenen GNU/Linux Communities. Dabei werden die Konstruktion der epistemischen Regime analysiert und normative Setzungen sowie die Rolle von Expertise herausgearbeitet. Dies dient der Beantwortung der ersten Forschungsfrage: *Wie prägen geteilte normative Vorstellungen der Mitglieder spezifische epistemische Regime?*

Anschließend werde ich die Wirkung dieser Regime auf die Wissensproduktion zeigen, um die zweite Forschungsfrage zu beantworten: *Wie wirken Regime auf das gemeinsame Wissensprodukt?*²⁷

Wie im vorigen Kapitel beschrieben, betrachte ich drei Variablen, die epistemische Regime beschreiben:

- die Konstitution der *Mitgliedschaft* zur Community,
- die *Strukturen und Verfahren* der Communities und
- die Bewertung der *Beiträge* (vgl. Abb. 7.1).

Zur Erhebung der spezifischen Merkmale der epistemischen Regime der Communities ist ein Vorgehen nach Methoden der qualitativen Sozialforschung sinnvoll, um die notwendige Detailtiefe zu erreichen und die subjektiven Bedeutungszuschreibungen der Mitglieder zu erfassen. Die Rekonstruktion der Variablen kann am besten mit qualitativen Leitfaden-Interviews erreicht werden; so kann auf die Fragestellung fokussiert werden und dennoch wird ausreichend Offenheit sichergestellt, um die Bedeutungskontexte der Befragten zu berücksichtigen. Um die in der Praxis relevanten Momente zu identifizieren, habe ich die Durchführung von Experteninterviews gewählt. Als Expert*innen betrachte ich in diesem Zusammenhang Mitglieder aus den Communities. Das Sample setzt sich zusammen aus Mitgliedern verschiedener Bereiche der Communities (Entwickler*innen und User) mit unterschiedlichen Erfahrungsstufen, da diese jeweils unterschiedliche Perspektiven auf die Community haben.

²⁷ Zur Fragestellung siehe detailliert S. 23.

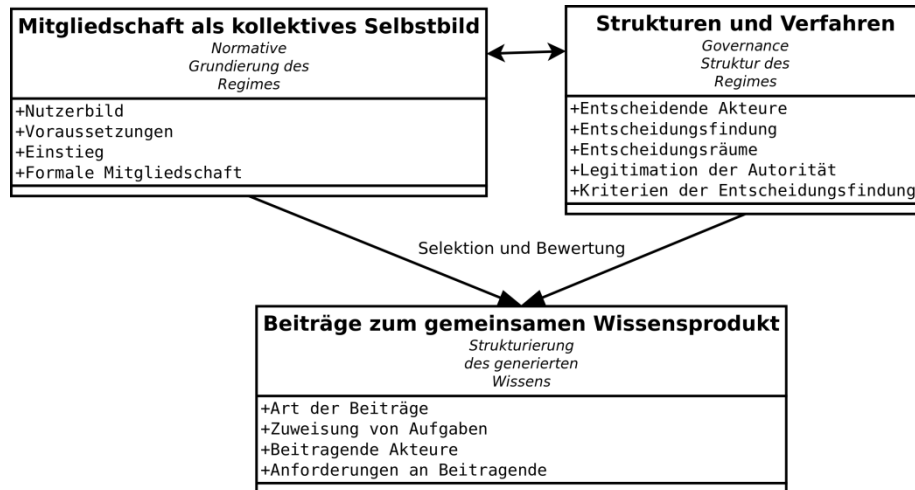


Abb. 7.1 Variablen epistemischer Regime

Eine Betrachtung der schriftlichen Kommunikate – in Form von Mailinglisten und Foren reichlich vorhanden – ermöglicht zwar die Gewinnung von Informationen über die Community und lässt auch Rückschlüsse zu über die Einstellungen und Erfahrungen der Community-Mitglieder, jedoch bleiben hier viele Details sehr implizit. In Interviews können die gewünschten Informationen hingegen direkter angesprochen werden und Nachfragen an die Interviewten können die Einblicke in bestimmte Aspekte vertiefen. Außerdem können die subjektiven Wertungen und Priorisierungen durch eine möglichst offene Gesprächsgestaltung erhoben und berücksichtigt werden. Der jeweilige situative und metasprachliche Kontext ist in schriftlichen Kommunikaten dagegen kaum ersichtlich, was die Aussagen schwer bewertbar macht. Verwendung finden die schriftlichen Kommunikate in meiner Arbeit allerdings ergänzend zur Validierung und punktuell zur Veranschaulichung der in den Interviews gewonnenen Informationen. Durch die Kenntnisse und Informationen der qualitativen Interviews können schließlich die schriftlichen Konversationen auch besser eingeordnet werden.

Zu Beginn des Gesprächs wurden die Interviewten nach ihrem *Einstieg* in die FLOSS-Community gefragt und danach, welche Projekte und Distributionen hier eine wichtige Rolle gespielt haben. Bei der Thematisierung von *Beitragsmöglichkeiten* innerhalb der jeweiligen Community wurde darauf geachtet, den Interviewten zunächst Freiraum für die Nennung von Beiträgen zu geben, ohne diese bezüglich Expertise und Bewertung schon durch die

Fragestellung vorzustrukturieren oder gar zu bewerten. Im Verlaufe des Gesprächs wurden durch explizites Nachfragen nach und nach die fehlenden Informationen gewonnen. In einem dritten Teil wurden die Interviewten nach Erfahrungen bezüglich *Entscheidungsprozessen* gefragt, um hierbei die Governance-Strukturen und Verfahren der Communities zu erheben.

Bezüglich der Governance-Strukturen der Communities gibt es zwar teils sehr detaillierte Beschreibungen auf Webseiten und in schriftlichen Dokumenten, aber wichtiger für die gelebte Praxis in der Community erscheinen insbesondere diejenigen Strukturen, die für die Mitglieder der Community aus eigener Erfahrung präsent und sichtbar sind. Die in der Praxis der Community relevanten Aspekte der Regeln und Strukturen der Communities wurden daher maßgeblich aus den Interviews erhoben und durch die schriftlichen Beschreibungen validiert und ergänzt.

7.1 Distributionen als Querschnitt verschiedener „Programmierkulturen“

Für die Fallauswahl ist es notwendig, die Varianz der betrachteten Fälle zu kontrollieren, insbesondere ist die Vergleichbarkeit unterschiedlicher FLOSS-Communities aus verschiedenen Gründen nicht trivial. Zum einen gibt es eine große Anzahl von Projekten mit sehr wenigen Beteiligten bis hin zu Eine-Person-Projekten, aber auch Mammut-Projekte, an denen Hunderte von Entwickler*innen mitarbeiten.

Zum anderen sind es aber auch unterschiedliche Eigenschaften verschiedener Arten von Software, die mit unterschiedlichen Wissenskulturen verbunden sind, was den Vergleich der Communities in Bezug auf die Rolle von Expertise sowie ihrer normativen Orientierungen erschwert.

7.1.1 „Programmierkulturen“

Zunächst möchte ich kurz ausführen, inwiefern sich die Wissenskulturen verschiedener Softwareprojekte unterscheiden:

Systemtiefe Konkret unterscheiden sich die einzelnen Programme darin, in welcher Ebene im System sie angesiedelt sind. Es gibt einerseits reine

Anwendungsprogramme wie beispielsweise Office-Anwendungen und andererseits systemnahe Programme wie bspw. Gerätetreiber – also Programme, die die Benutzung eines Geräts für andere Programme und die User steuern.

Anwendungsdomains Dabei unterscheiden sich auch die *Kreise der Anwender*innen* und demzufolge spielen hier unterschiedliche Arten von Wissen bezüglich der Funktionsweise der Programme und der Anwendungsdomain eine Rolle. Ein Programm zur Netzwerkadministration erfordert beispielsweise andere Kenntnisse über die Anwendung des Programms und der darunter liegenden funktionalen Zusammenhänge als Programme zur Steuerung der Grafik. Statistikprogramme erfordern anderes Wissen als Textverarbeitungsprogramme. Dies gilt sowohl für die Anwendung als auch für die Programmierung einer solchen Software. Dies wirkt sich aus auf die unterschiedlichen Arten von Wissen und Expertise, die sowohl bei der Nutzung als auch bei der Entwicklung entsprechender Programme notwendig sind.

Spezifische Umsetzung des Programms Aber selbst innerhalb einer Kategorie von Programmen wie zum Beispiel Textverarbeitung variieren die Wissenskulturen. Beispielsweise ist LaTeX ein Textverarbeitungsprogramm, bei dem die Formatierung des Textes über Befehle innerhalb des Text-Editors geschieht (ähnlich wie etwa bei HTML), Libre- und Open-Office hingegen sind wie Microsoft Word Programme, die den Text grafisch als fertige (Druck-)Seite darstellen („What You See Is What You Get“). Die Benutzung dieser unterschiedlichen Ansätze der Textformatierung impliziert also wiederum unterschiedliche Wissenskulturen der User und Entwickler*innen.

Sprachkultur Schließlich unterscheiden sich auch die Programmiersprachen, mit denen die Programme realisiert werden. Programmiersprachen unterscheiden sich nicht nur in ihrer Syntax (Struktur, Funktionsnamen und Zeichen), sondern auch in der Art oder Tiefe des Eingriffs in das System: ob beispielsweise der Arbeitsspeicher von dem oder der Programmier*in selbst verwaltet werden muss und auf welche Weise oder ob das von einer Zwischenschicht erledigt wird – oder der logischen Herangehensweise, Probleme zu lösen (z.B. objektorientierte vs. funktionale Programmiersprachen). Damit können sich zwei funktional äquivalente Programme wesentlich darin unterscheiden, welche Kenntnisse der technischen Architektur des Computers notwendig sind, um das Programm in seiner Funktionsweise zu verstehen.

Entwicklung als Form der Anwendung Eine weitere Schwierigkeit der Differenzierung von Wissenskulturen ergibt sich schließlich bei FLOSS-Projekten, deren gemeinsames Produkt eine Programmiersprache selbst ist, in

7.1 Distributionen als Querschnitt verschiedener „Programmierkulturen“ 105

denen also die User der Software (der Programmiersprache) Entwickler*innen anderer Software sind.

7.1.2 Charakterisierung von Distributionen

In der vorliegenden Arbeit wurden daher eine besondere Art von Communities zur Betrachtung gewählt, sogenannte *GNU/Linux-Distributionen*.²⁸ Diese stellen Programme aus dem verfügbaren Pool an FLOSS-Software-Projekten zusammen und bieten diese in Kombination mit den für ein Betriebssystem notwendigen Systemtreibern und Konfigurationswerkzeugen an (vgl. Abb. 7.2).

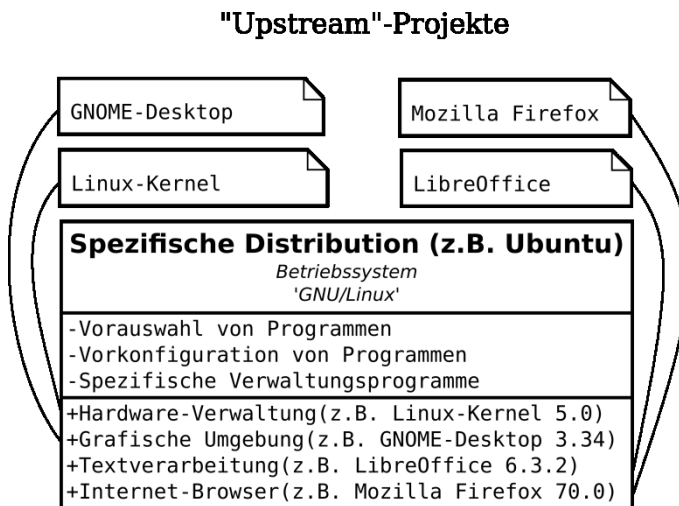


Abb. 7.2 Schematische Darstellung einer Distribution

Somit liegen Distributionen gewissermaßen quer zu den einzelnen Projekten: Durch die Integration von Anwendung und Systemebene fließen hier Oberflächen-Anwendungen, Programmiersprachen bis hin zu Hardware-Treibern zusammen und die Softwareentwicklung in der Community deckt eine Breite von Funktionalitäten und Wissensgebieten ab. In den Distributionen finden sich Entwickler*innen aus den verschiedenen Projekten (und den unterschiedlichen Programmier-Kulturen) zusammen und einigen sich auf

²⁸ Üblicherweise wird das Betriebssystem schlicht Linux genannt, dabei steht der Name „Linux“ genau genommen nur für den Kernprozess der Distributionen, während die Funktions-Programme drumherum großteils aus dem GNU-Projekt stammen.

die harmonische Gestaltung eines Betriebssystems, in dem die Einzelteile funktional zusammengefügt sind.

Über die Integration und Abstimmung der einzelnen Software-Programme hinaus entwickeln die Distributions-Communities eigene Programme zur Konfiguration und Installation des Systems und zur Verwaltung und Aktualisierung der installierten Programme. Als umfassendes *Betriebssystem* ist die Distribution schließlich die allgegenwärtige Schnittstelle zwischen User und Hardware und eignet sich ideal für die Betrachtung des Verhältnisses von Expert*innen und Laien in Bezug auf die Nutzung der spezifischen Technik *Computer*.

Upstream und Downstream

Im Jargon der Communities wird eine Fluss-Metapher verwendet, um zu unterscheiden zwischen den originären Software-Projekten und ihren Modifikationen. Aus Sicht der Distribution kommen die mannigfaltigen Programme von „flussaufwärts“, dem sogenannten *Upstream*. Auf der Ebene der Distribution werden üblicherweise spezifische Anpassungen vorgenommen, um das Zusammenspiel der einzelnen Software-Komponenten in einem harmonisch funktionierenden Betriebssystem zu verbessern – sie liegt demnach „flussabwärts“, *downstream*, vom Ursprungsprojekt. Die Upstream-Projekte werden von eigenen Communities gepflegt und weiterentwickelt. Die Unterscheidung in Upstream und Downstream hat eine besondere Relevanz beim Melden und Beheben von Fehlern. Da die Software auf Downstream-Ebene verändert wird, ist es wichtig zu klären, ob die Fehler auch im Upstream existieren. Ist das der Fall, ist es wiederum wichtig, dass die Fehler auch dort korrigiert werden, sonst fließen mit neueren Versionen die Fehler wieder abwärts und müssen erneut behoben werden.

Die Anpassungen auf der Ebene der Distribution haben einerseits zum Ziel, die Software nach den Vorlieben der jeweiligen Community zu konfigurieren. Andererseits wird sichergestellt, dass verschiedene Software-Programme miteinander reibungslos funktionieren. Durch die ständige Weiterentwicklung der Software kommt es hin und wieder zu Veränderungen der Funktionalität und der Kompatibilität mit anderen Programmen, daher pflegen die Distributionen „flussabwärts“ teils ältere Versionen von Programmen, speisen aber sicherheitskritische Änderungen der neueren Versionen mit ein.

Diese Anpassungen sind mit zusätzlicher Arbeit auf der Ebene der Distribution verbunden. Dazu muss manchmal mehr und manchmal weniger stark

7.1 Distributionen als Querschnitt verschiedener „Programmierkulturen“ 107

in den Softwarecode eingegriffen werden. Ein Verständnis der Upstream-Software sowie der damit verbundenen Abhängigkeiten ist also auf der Ebene der Distribution notwendig. Häufig gibt es enge Kooperationen (oder auch Personalunionen) zwischen Upstream-Entwickler*innen und den Betreiber*innen der entsprechenden Downstream-Software-Pakete²⁹ in den Distributionen.

Eignung von Distributionen für den Fallvergleich

Innerhalb einer Distribution werden Upstream-Programme aufeinander abgestimmt und vorkonfiguriert, um den Nutzer*innen einer Distribution eine funktional harmonisierende Softwareumgebung zur Verfügung zu stellen, mit der sie die Hardware (also den Computer) sinnvoll nutzen und die angestrebten Aufgaben erledigen können. Das Ergebnis ist ein Betriebssystem mit Programmen für alle möglichen Aufgaben, die Computer-Nutzer*innen erledigen möchten, vom Foto-Drucker über die Online-Buchung bis hin zum E-Mail-Versand. Diese sind teils vorinstalliert und ansonsten in einer Softwareverwaltung verfügbar, in der Software-Pakete bereitgestellt werden. Darum steht bei der GNU/Linux-Distribution als Betriebssystem insbesondere die Nutzbarmachung der Technik Computer als Ressource des Handelns der User im Mittelpunkt und eignet sich daher für die Analyse der Rolle von Expertenwissen und der Betrachtung einer Wissensdifferenz zwischen Expert*innen und Laien.

In Distributions-Communities beteiligen sich verschiedene Entwickler*innen und User, die Teil verschiedener Programmier- und Nutzerkulturen sind. In den Distributions-Communities finden sich auch Nutzer*innen wieder, die sich auf unterschiedliche Art und Weise (upstream wie downstream) beteiligen. Gemeinsam ist den Mitgliedern solch einer Community ein gewisser Konsens darüber, wie ein Betriebssystem auszusehen hat, was die architektonische Bauweise, das Standard-Set an Programmen und die Standard-Konfiguration angeht sowie die Beschaffenheit der Administrationswerkzeuge

²⁹ Der Begriff des Pakets beinhaltet dabei die Menge der für die Ausführung notwendigen Dateien. Dabei gibt es aufgrund des modularen Aufbaus der einzelnen Projekte Programmelemente, die mit anderen Softwareprogrammen geteilt werden. Daraus resultieren wiederum Abhängigkeiten zwischen Programmen und deren Bibliotheken, die über die Paketverwaltung überwacht und verwaltet werden. In ihrer Erscheinung für die User ist die Paketverwaltung ähnlich zu sogenannten App-Stores, die sich in den letzten Jahren für mobile Betriebssysteme und schließlich auch für MS Windows etabliert haben.

(etwa Assistenzprogramme zur Treiber-Installation). Ausgehend von der Standard-Konfiguration sind aber vielerlei Anpassungsmöglichkeiten vorhanden.³⁰ Somit findet sich in einer Distributions-Community ein Querschnitt verschiedener Programmkulturen (horizontal) sowie unterschiedlich technisch versierte User (vertikal) wieder, was die Vergleichbarkeit der Fälle für die durchgeführte Analyse gewährleistet.

Gemeinsame Werte der verschiedenen Communities zeigen sich in ihrer jeweiligen Standard-Konfiguration – sozusagen dem Auslieferungszustand des Komplet-Systems, das sich durch je unterschiedliche Software-Auswahl sowie verschiedenartige Administrations-Werkzeuge auszeichnet. Parallel zu der Art der Software-Gestaltung zeigen sich aber auch abweichende Wertssysteme in der Gestaltung und Nutzung der Kommunikations- und Kollaborationswerkzeuge sowie in den informellen Praktiken und der formalen Organisation der Software- und Wissensproduktion. Eine vergleichende Analyse dieser verschiedenen Aspekte ist daher geeignet, die normativen Prägungen der epistemischen Regime der Communities und der Beziehungen zwischen sozialer Struktur und Wissensprodukt zu untersuchen.

7.1.3 Auswahl und Beschreibung der untersuchten Distributionen

Zur vergleichenden Betrachtung werden drei verschiedene GNU/Linux-Distributionen herangezogen, die im Folgenden näher beschrieben werden. Ein naheliegender Untersuchungsfall war die Distribution *Debian Linux*, einerseits, weil der Fall schon in der Literatur diskutiert wurde (vgl. bspw. Coleman 2013; Lazaro 2008) und somit Anknüpfungspunkte für die wissenschaftliche Debatte liefert, und andererseits, weil er einen Wendepunkt darstellt von der arbiträren Oligarchie der Core-Entwickler*innen hin zu einer bürokratischen Organisation der Community (vgl. Coleman 2013: 129 f.). Die Entstehung von Debian ist eine Innovation im Hinblick auf die Entwicklung eines Regimes, das auf Transparenz, demokratische Entscheidungsfindung und Legitimation setzt (vgl. ausführlich O'Mahony/Ferraro 2007: 1088 ff.).

30 Insbesondere gibt es eine Vielzahl an grafischen Oberflächen, die komplett austauschbar sind und unterschiedliches „Look & Feel“ bieten; dies ist unabhängig von der Standard-Konfiguration frei konfigurierbar. Es gibt also weitestgehende Wahlfreiheit, aber bestimmte Setzungen sind dennoch durch die *Vorkonfiguration*, die Setzung der *Standard-einstellungen*, gegeben.

7.1 Distributionen als Querschnitt verschiedener „Programmierkulturen“ 109

Vor dem Hintergrund der Fokussierung auf die Rolle von Expertise in den sozialen Strukturen der Communities und in ihren Selbstverständnissen ist es sinnvoll, in dieser Hinsicht eine klare Varianz der Auswahl der Fälle zu konstruieren. Die Fallauswahl und kurze Charakterisierungen der einzelnen Fälle werde ich im Folgenden erläutern.

Ubuntu Linux stellt einen weiteren Meilenstein der Entwicklung der GNU/Linux-Landschaft dar. In besonderer Weise wurden hier explizit die Laien-User adressiert und dieses auch erfolgreich vermarktet: „Linux for Human Beings“. Darüber hinaus wurde hier die Hegemonie des De-facto-Standards Microsoft Windows explizit als zu behebender Fehler deklariert und damit das Ziel formuliert, die Nutzergruppe „Normalverbraucher“ zu erreichen.³¹ Nicht zuletzt mithilfe einer guten Ausstattung an finanziellen Ressourcen für die Bezahlung von Entwickler*innen, Designer*innen und Marketing durch den Gründer der Firma Canonical wurde das Projekt sehr schnell bekannt und erfolgreich,³² gerade auch in der Ansprache von Umsteigern von Microsoft Windows, und etablierte sich zur Standard Einsteigerdistribution.³³

Während *Debian Linux* eher als konservative Distribution gilt, die einen besonderen Wert auf Stabilität und den Fokus auf Software mit freien Lizenzen legt, ging Ubuntu von Anfang an einen pragmatischeren Weg in der stärkeren Integration von proprietärer Software. Während bei Debian der Leitsatz gilt, die neue Version ist fertig, wenn sie fertig ist – was manchmal dann eben etwas länger dauert –, kündigte Ubuntu von Anfang an einen halbjährlichen Release-Zyklus an. Interessant für einen Vergleich ist dabei auch die Tatsache, dass Ubuntu als Ausgangspunkt ihrer neuen Releases den Entwicklungszweig von Debian nutzt, und somit darauf basiert. Dazu als Erläuterung:

31 <https://bugs.launchpad.net/ubuntu/+bug/1>, letzter Aufruf 4.5.2017

32 Auf dem Popularitätsindex der Seite distrowatch.com war Ubuntu im Jahr nach der Gründung (2005) bereits auf Platz 1, und blieb seither in den Top 10. Aktuell listet die Seite über 200 Distributionen. Der Popularitätsindex misst jedoch nur die Aufrufe der Beschreibungsseite auf DistroWatch und sagt nichts aus über die tatsächliche Nutzung der Distributionen, dennoch gibt das Ranking Hinweise über das öffentliche Interesse an den gelisteten Distributionen.

33 Möglicherweise spielte Ubuntu in der Ansprache von Windows-Usern auch deren Unzufriedenheit mit den nicht ganz ausgereiften Versionen *Millenium Edition* und *Vista* bzw. die Gerüchte und Verzögerungen um *Longhorn* (der damalige Codename von *Vista*) in die Hände.

Debian bietet neben einer *stable* Version auch einen *unstable* Zweig, in dem die laufenden Änderungen stattfinden, die erst nach weiterer Überprüfung und einer Bewährungsphase Eingang in die stabile Distribution finden.³⁴ Technisch gibt es also zunächst eine starke Ähnlichkeit der Software-Pakete, während die Wertsetzungen (Stabilität vs. Usability) und die Organisationsformen sich stark unterscheiden und auch in den Paketen die Upstream-Pakete aus Debian im Downstream-Ubuntu in mancher Hinsicht geändert sind. So zeigen sich in der Gestaltung der Software und durch die spezifischen *Konfigurationen* deutliche Unterschiede, wie ich im Kapitel 9 herausarbeiten werde.

Wohlgermerkt wurde Ubuntu zwar von der Firma Canonical initiiert und es gibt einige bezahlte Entwickler*innen und Angestellte, die an Ubuntu mitarbeiten, aber parallel arbeiten auch viele Entwickler freiwillig an der Software. Der Sponsor und Inhaber von Canonical, Mark Shuttleworth, steht zwar an der Spitze, nimmt aber erklärtermaßen nur dort eine Führungsrolle ein, wo kein mehrheitlicher Konsens erreicht werden kann. Für Entwicklung, Dokumentation und Support spielt also dennoch die Community eine zentrale Rolle, sodass eine Vergleichbarkeit mit den anderen Fällen gegeben ist (mehr dazu im folgenden Kap.).

Schließlich liegt der Untersuchung ein dritter Fall zugrunde, der eine explizite Ausrichtung an erfahrene oder zumindest lernwillige User repräsentiert und sich darüber hinaus kritisch gegenüber dem Paradigma der Usability positioniert. Durch diese explizite Abgrenzung von Laien stellt dieser Fall einen interessanten Kontrastfall in Bezug auf die Gestaltung der Experten-Laien-Differenz dar. Man kann sagen, in *Arch Linux* herrscht das Credo, dass das Verstehen der Funktionsweise der Technik für die sinnvolle Verwendung des Systems notwendig ist, um als User aufkommende Probleme selbst lösen und aktiv zur Entwicklung des Projekts beitragen zu können. Arch ist weder von einer Firma geführt noch gibt es eine bürokratische Ausdifferenzierung wie bei Debian. Arch ist zwar ein deutlich kleineres Projekt als die beiden Vergleichsfälle, ist aber wie die beiden anderen sehr populär³⁵ und wohl das

34 Genauer gesagt fließen die *unstable*-Pakete nach einer kurzen Bewährungszeit in die Version (besser: den Distributionskanal) *testing*. Im Abstand von ca. zwei Jahren wird dann *testing* nochmal ausgiebig überarbeitet und dann in das nächste *stable* Release überführt.

35 Auf dem Popularitäts-Index von DistroWatch ist Arch momentan auf Platz 12 von über 200 gelisteten Distributionen

7.1 Distributionen als Querschnitt verschiedener „Programmierkulturen“ 111

populärste Linux mit einer derart explizit technischen Ausrichtung. Wie Debian und Ubuntu gibt es auch von Arch mittlerweile abgeleitete Distributionen, was deren Relevanz unterstreicht, da abgeleitete Distributionen (Downstream) immer auf der Upstream-Distribution aufbauen.³⁶

Die Fälle lassen sich wie folgt zusammenfassen:

Debian (gegründet 1993) ist das Linux mit der vermutlich größten Anzahl an unterstützten unterschiedlichen Rechnerarchitekturen (unterschiedliche Prozessoren) bei einer großen Anzahl von zur Verfügung gestellten Software-Programmen („Pakete“). Besonders hervorzuheben ist, dass diese Distribution vollständig von einer Community betrieben wird und eine verfasste demokratische Struktur hat. Außerdem hat die Community eigens Richtlinien für die Akzeptanz von Software als „Freie Software“ entwickelt, die *Debian Free Software Guidelines* (DFSG). Mit dem Claim „the universal operating system“ adressiert die Distribution kompetente Nutzer, die bei der Betreuung mehrerer Computer einen gewissen Administrations-Komfort schätzen.

Ubuntu wurde 2004 vom Millionär und Weltraumtouristen Mark Shuttleworth ins Leben gerufen, um Linux für die breite Masse attraktiv zu machen – und nicht zuletzt, um damit die Monopolstellung von Microsoft zu unterwandern. Die Distribution erfreute sich bald sehr großer Beliebtheit und setzt nach wie vor Standards in Sachen Nutzerfreundlichkeit. Der Slogan „Linux for human beings“ steht insbesondere für die Adressierung von Umsteigern und Anfängern, wenngleich Ubuntu auch eine Variante für den Serverbetrieb anbietet. Im Hintergrund des Projektes steht weiterhin Mark Shuttleworth als Sponsor und Inhaber der Firma Canonical, die bezahlte Angestellte für Ubuntu beschäftigt.

Arch wurde 2001 gestartet und folgt dem Prinzip der Einfachheit, das auch „Keep it simple, stupid“ genannt wird (KISS). Hiermit ist im Gegensatz zur einfachen Bedienbarkeit von Ubuntu eine technische Einfachheit gemeint im Sinne einer möglichst nachvollziehbaren technischen Funktionsweise: „A simple, lightweight distribution“. Die Distribution möchte den Nutzer*innen die volle Kontrolle über das System geben, damit haben diese auch eine gewisse Verantwortung, die mit notwendigen Kompetenzen einhergeht.

³⁶ Ein anschaulicher Linux-Stammbaum findet sich hier: https://upload.wikimedia.org/wikipedia/commons/1/1b/Linux_Distribution_Timeline.svg, letzter Aufruf 4.5.2017

7.2 Sampling

Als sinnvolle Interviewpartner*innen – also „Expert*innen“ ihrer Domäne im Sinne von Experten-Interviews in Abgrenzung zum hier ansonsten verwendeten Begriff technischer Expertise – betrachte ich Mitglieder der Community. Um verschiedene Perspektiven bezüglich der technischen Expertise zu erfassen – nun im Sinne eines Expertensystems, also der Perspektive derer, die die Funktionsweise der Technik verstehen und verändern können –,³⁷ war es für das Sampling der Interviews wichtig, sowohl Entwickler*innen als auch User, die keine Entwickler*innen sind, aufzunehmen. Darüber hinaus mussten aus der Gruppe der Entwickler*innen und der User jeweils Mitglieder der verschiedenen Communities vertreten sein, um für alle Fälle die notwendigen Perspektiven zu erhalten.

Die Mitgliedschaft als eine Form der Selbstzuschreibung und *gefühlten* Zugehörigkeit ist nicht immer von vornherein uneindeutig. Im Falle von Entwickler*innen ist die Zugehörigkeit zur Community recht klar durch wiederkehrende Beiträge zum Softwarecode zu erkennen. Im Falle von Usern diente mir eine über die reine Nutzung hinausgehende Involvierung in die Community als Indiz für eine gewisse Auseinandersetzung mit der Community und den entsprechenden Wissensstrukturen. Dies zeigte sich beispielsweise in Form von Linux-Support im Bekanntenkreis oder gar die Teilnahme an Open-Source-Software-Konferenzen (auf denen ich einige Interviews durchgeführt habe). Darüber hinaus wurde in den Interviews die Nutzung und Zugehörigkeit zu Linux-Distributionen auch thematisiert, um die Aussagen einordnen zu können.

Ein Großteil der Interviews wurde auf Konferenzen durchgeführt. Teils wurden Vortragende vorab aufgrund ihrer Selbstdarstellungen auf der Konferenz-Website angesprochen, teils wurden Teilnehmende angesprochen und in einem kurzen Gespräch erwogen, ob ein Interview zielführend sein könnte. Teils konnten auch die Interviewten geeignete Gesprächspartner*innen für eine Befragung vermitteln. Das Sampling wurde nach und nach ergänzt, um fehlende Punkte im Sampling abzudecken und eine ausreichende Dichte an Informationen zu gewinnen. So war es möglich, auch in den zeitlichen Zwischenräumen zwischen den Interviews weiteren Informationsbedarf festzu-

³⁷ Zur Konzeption von Software als Expertensystem siehe Kap. 4.

stellen und diesbezüglich auch gezieltere Fragen in spätere Interviews einfließen zu lassen.

Der Auswertung wurden schließlich 16 Interviews zugrunde gelegt, die jeweils ca. 30 bis 45 Minuten dauerten. Zwei davon waren Gruppen-Interviews, eines mit drei *Arch Developern* (Interview 8) und eines mit einem *Debian Developer* und seinem Kollegen (Interview 2), der mit ihm an einem anderen FLOSS-Projekt mitentwickelt und mit dem Feld vertraut ist.

Einige der Interviewten entwickeln für unterschiedliche Distributionen. Durch die Verwandtschaft von Debian und Ubuntu tragen Entwickler*innen von Ubuntu teils auch in Debian bei, um die Bugs in Ubuntu auch oder direkt in den Upstream (Debian) einzupflegen.

Üblicherweise sind die Entwickler*innen einer Distribution auch User der Distribution. Wenn sie bei mehreren Distributionen eine Entwickler-Rolle innehaben, haben die Interviewten meist eine präferierte Distribution, die sie maßgeblich nutzen, oder aber sie nutzen verschiedene Distributionen für verschiedene Zwecke. In der Regel fangen sie an, beizutragen, weil sie es nutzen (12,7)³⁸ oder hören auf, beizutragen, wenn sie es nicht mehr nutzen (11,32). Besonders klar ist diese Überschneidung bei Arch, wo sich die Entwickler*innen stufenweise aus den aktiven Usern und den sogenannten *Trusted Usern* rekrutieren (s. Abschnitt 8.2.3).

Einige Interviewte sind reine Nutzer*innen, die keine Entwickler-Rolle innehaben, dies war auch Ziel des Sampling. Dabei stechen zwei User hervor, die unterschiedliche Distributionen nutzen – auch die hier betrachteten Fälle –, jeweils für verschiedene Zwecke. Ein solcher Mutli-User hatte einen professionellen Hintergrund (Interview 10), der andere nutzte die Distributionen eher in seiner Freizeit (Interview 13).

Da die meisten Interviewten also mehrere Rollen innehaben – weil sie entweder in verschiedene Communities involviert oder neben der Entwicklung auch User sind –, ergeben sich für die insgesamt 18 interviewten Linux User/Developer³⁹ folgende Mitgliedschaften und Rollen:

38 Referenzen zu Interview-Stellen werden durch die Nummer des Interviews, und der Zeilennummer nach dem Komma in Klammern belegt. Hier also Interview Nummer 12, Zeile 7.

39 Einer der eben aufgeführten Interviewten war eben kein Linux-User/Developer.

Tab. 7.1: Rollen der Interviewten

Rolle	Ubuntu	Debian	Arch
Developer	5	6	4
User	10	6	7
„Nur“-User	5	2	3

Mit den verschiedenen Rollen gehen für die Fragestellung relevante Einsichten und Erfahrungen in die epistemischen Regime der Communities einher.

Auf eine Gender-Balance wurde kein besonderes Augenmerk gelegt, da das Geschlecht für die Herausarbeitung der epistemischen Regime eine nachrangige Rolle spielen sollte. Generell sind Frauen in FLOSS-Communities aber stark unterrepräsentiert, was in den Communities teils als Problem gesehen wird, zudem ist die Gender-Balance auch Gegenstand jüngerer Studien (vgl. Kuechler/Gilbertson/Jensen 2012). Interessanterweise gibt es auch Projekte, die explizit die Beteiligung von Frauen unterstützen.⁴⁰ Die weibliche Beteiligung wurde zwischen 2% in älteren Studien (vgl. Robles/Scheider u. a. 2001) und 10% in neueren Studien (vgl. Robles/Reina u. a. 2016; Vasilescu u. a. 2015) berechnet. Mit 1/18 weiblichen Interviewten liegt der Anteil der Befragten in der vorliegenden Studie immerhin bei rund 6%.

Über die Interviews hinaus wurden Informationen auf verschiedenen Entwicklerkonferenzen und Linux-Veranstaltungen in Gesprächen und Vorträgen gewonnen, diese Informationen beeinflussten nicht zuletzt die Auswahl der weiteren Interviewpartner*innen und die Auswahl der auszuwertenden Interviews. Insbesondere besuchte ich die „Free and Open Source DEveloper Meetings“ (FOSDEM) 2015 und 2016 in Brüssel, die „Free and Open Source Convention“ (FROSCON) 2014 in Bonn sowie die „Chemnitzer Linux-Tage“ 2016. Darüber hinaus besuchte ich regelmäßig eine Linux-User-Gruppe, in deren Rahmen ich mich auch aktiv an Linux-Install-Partys und ähnlichen Veranstaltungen beteiligte. Nicht zuletzt durch eigene Erfahrungen mit unterschiedlichen Anwendungs-Kontexten und Tätigkeiten in der Systemadministration und Software-Entwicklung habe ich mir über viele Jahre eine fundierte Kenntnis des Feldes angeeignet.

40 Beispielsweise seien hier „Debian Women“, <https://www.debian.org/women/>, und die „Rails-Girls“, <http://railsgirls.com>, genannt.

Parallel zur Interview-Auswertung wurden punktuell die User- und Entwickler-Mailinglisten der betrachteten Fälle untersucht. Mit verschiedenen Begriffen habe ich nach einer Heuristik gesucht, um systematisch für die Fragestellung relevante Konversationen zu finden. Es stellte sich aber heraus, dass die Text-Konversationen keine reichhaltigen zusätzlichen Erkenntnisse für die spezifische Beantwortung der Forschungsfrage ergaben. Allerdings brachte die Suche beispielhafte Korrespondenzen hervor, die ich an geeigneter Stelle verwende, um die beschriebenen Sachverhalte zu validieren und zu veranschaulichen.

Für die Analyse in Kapitel 9 habe ich außerdem autoethnografisch Linux-Installationen durchgeführt, wobei ich hierbei auf langjährige Erfahrungen, auch im Rahmen der Unterstützung Dritter bei Linux-Installationen, zurückgreifen konnte.

7.3 Auswertungsmethode

Die Experteninterviews wurden transkribiert und anonymisiert. Die Themen sind zwar grundsätzlich unkritisch, aber um die größtmögliche Offenheit seitens der Interviewten zu erreichen, wurde dennoch eine Anonymisierung der Interviews vereinbart. Die Interviewten waren grundsätzlich aufgeschlossen und interessiert. Das Teilen von Wissen entspricht dem Leitgedanken der FLOSS-Community und eine Beschäftigung mit deren Idealen und Praktiken wird tendenziell wohlwollend aufgenommen.

Für die Auswertung des Materials folgte ich dem Ansatz der qualitativen Inhaltsanalyse nach Gläser und Laudel (2008). Im Unterschied zu Kodierverfahren, die oft dazu verwendet werden, um Narrative zu markieren und darauf basierend Theorien aus dem empirischen Material abzuleiten (Grounded Theory), ist die Inhaltsanalyse dazu geeignet, theoretisch informiert relevante Informationen zu extrahieren, um offene Fragen der Theorie zu adressieren. Aus den theoretischen Vorüberlegungen wurden dafür in Abschnitt 6.3 Untersuchungsvariablen und dazugehörige relevante Dimensionen erarbeitet, um ein theoretisches Selektions- und Analyseraster zu gewinnen.

Die Verwendung von Extraktionstabellen ermöglicht nicht nur, Textstellen mit Schlagworten zu indexieren (wie beim Kodieren), sondern zu den betreffenden Kategorien, d.h. den Variablen und ihren Dimensionen, die

extrahierten Informationen direkt für jede Textstelle in die Tabelle einzutragen. Zur besseren Nachvollziehbarkeit und für die Möglichkeit, bei Bedarf wieder ins Material zu gehen, werden die Textstellen in der Tabelle referenziert.

Bei unerwarteten Funden wird die Tabelle gegebenenfalls um fehlende Dimensionen erweitert; möglich ist grundsätzlich auch, dass sich in der Auseinandersetzung mit dem Material eine weitere Variable für das Modell ergibt und Variablen und Dimensionen angepasst werden. Somit wird dem Prinzip Offenheit Rechnung getragen.

Anstelle der Verwendung des MS-Word-Skripts MIA von Gläser und Laudel (ebd.) nutzte ich für die Analyse der Interviews einen Texteditor. Die Extraktionstabellen erstellte ich in einem (FLOSS-)Tabellen-Programm, anstatt sie aus dem MIA-Skript heraus zu generieren.⁴¹

Je Variable bildet eine Extraktionstabelle in Spalten die Dimensionen der Variablen ab. Für jede gefundene Textstelle wurden zunächst die relevanten Informationen in eine Zeile eingetragen und mit einer Referenz zur Textstelle im Interview versehen, um sie auch später noch zuordnen zu können.⁴² Während der Auswertung war es möglich, im Laufe des Erkenntnisfortschritts weitere Spalten zu ergänzen, um nicht vorhergesehene Dimensionen aufzunehmen. Eine zusätzliche Spalte diente als Memo sowie eine gesonderte Tabelle, Beobachtungen aufzunehmen, die zunächst nicht unmittelbar den Variablen zuzuordnen waren.

Im Laufe der Studie kristallisierte sich die Variable „Mitgliedschaft als Selbstbild der Community“ aus einer anfänglich angenommenen Kategorie

41 Somit ersparte ich mir die Übersetzung des MIA-Visual-Basic-Skripts von MS Word nach Libre Office und konnte dennoch recht komfortabel die Auswertungsmethode nutzen. Ein anderer Weg zur Realisierung eines Tools stellt meines Erachtens die Weiterentwicklung bestehender FLOSS-Kodierprogramme dar. Ein relativ niederschwelliger Ansatz scheint mir hier zu sein, das in Java geschriebene FreeQDA zu ergänzen. Hier könnte man das XML-Datenmodell erweitern, um die Generierung von Extraktionstabellen zu ermöglichen. Leider war es mir aus Zeitgründen nicht vergönnt, mich hinreichend tief in die Programmstruktur einzuarbeiten, beziehungsweise war der Workaround mit Texteditor und Tabellenprogramm im Vergleich dazu der deutlich schnellere Weg. FreeQDA ist eines von vielen Kodierprogrammen im Beta-Stadium. Mehr dazu siehe <http://www.produnis.de/fqda/> oder <https://github.com/produnis/FreeQDA> (letzter Aufruf 4.5.2017).

42 In den folgenden Kapiteln bezeichnen folglich die Belege in Klammern die Nummer des Interviews, nach dem Komma die Zeilennummer.

„Nutzerkultur“ heraus. Die Dimension der „Zuweisung von Aufgaben“ hatte ich zunächst dort verortet, es ergab sich aber aus dem Material, dass die Zuordnung zur Variable „Beitrag“ sinnvoller ist. Eine weitere Extraktions-Kategorie „Rollen“ hatte ich anfangs als zentral erachtet. Die Analyse ergab aber, dass deren Bedeutung sich anderen Variablen unterordnet, weshalb sie sich im Wesentlichen in „Strukturen und Verfahren“ auflöste.⁴³ Es wird also deutlich, dass trotz der theoretischen Informierung Offenheit gegenüber den empirischen Bedeutungszusammenhängen gegeben ist, um die ursprüngliche Konzeption reflexiv anzupassen.

In einem weiteren Schritt wurden die Tabellen sortiert und zusammengefasst, um die Informationen zu bündeln und den Vergleich zu erleichtern. Dabei wurde durch die Aggregation und Gegenüberstellung der Interviewaussagen validiert oder – wenn eine Validation nicht möglich war – kritisch bewertet; oder es wurden weitere Materialien (beispielsweise Dokumentationen und schriftliche Konversationen aus Mailinglisten) zu Hilfe genommen.

Widersprüchlichkeiten zeigten sich in der Auffassung der Rolle der Firma Canonical und ihrer Finanzierung und ihrer damit verbundenen Führungsrolle. Die Rolle von Canonical und dem Gründer Mark Shuttleworth wird in Abschnitt 8.2.1 diskutiert. Die prägende Bedeutung der Rolle des Gründers für die Konfiguration des epistemischen Regimes wurde auch in teils konträren Aussagen deutlich.

Die Ergründung der Details der Geldflüsse zwischen Mark Shuttleworth, der Ubuntu Foundation und Canonical übersteigt den Rahmen meiner Arbeit. Für meine Analyse relevant ist aber die Bedeutung finanzieller Ressourcen in Bezug auf bezahlte Angestellte und ihre Rolle in der Community. Dies ließ sich eindeutig herausarbeiten und floss in die Analyse mit ein. So bilden die beobachteten Widersprüche einen Ansatz für weitere Forschung, können aber für mein Forschungsinteresse vernachlässigt werden.

In den folgenden Kapiteln werde ich nun die Untersuchung der dargestellten Fälle ausarbeiten und mit der erarbeiteten, theoretisch informierten Perspektive die Wirkungsweise des Zusammenspiels von epistemischen Praktiken und sozialer Ordnung als epistemische Regime analysieren. Anschließend beziehe ich die Ergebnisse auf meine Fragestellung, um die Fragen nach den Wechselwirkungen zwischen den normativen Vorstellungen der Mitglieder, der sozialen Struktur der Community und dem gemeinsamen Wissensprodukt zu beantworten.

43 Eine Dimension ging auf in die Mitglieds-Dimension „formale Mitgliedschaft“.

8 Epistemische Regime der gemeinschaftlichen Softwareproduktion

Nachdem ich die Fallauswahl und das methodische Vorgehen vorgestellt habe, widme ich mich nun der Auswertung des Materials und der Ausarbeitung der Ergebnisse. Basierend auf den von mir geführten Interviews im Feld werde ich anhand der in Kapitel 6 erarbeiteten Kategorien die unterschiedlichen epistemischen Regime von GNU/Linux-Communities beschreiben. Ergänzend greife ich, zur Veranschaulichung der Befunde, zurück auf Dokumente der Selbstbeschreibung der Communities und punktuell auf Konversationen aus den entsprechenden Mailinglisten.⁴⁴ Dabei geht es mir im Wesentlichen darum, die Unterschiede der verglichenen Communities herauszuarbeiten.⁴⁵

Entlang der drei bestimmenden Kategorien *Mitgliedschaft als kollektives Selbstbild, Strukturen und Verfahren* und *Beiträge zum gemeinsamen Wissensprodukt* vergleiche ich die Ausprägungen der Dimensionen der betrachteten Fälle und fasse das Ergebnis anschließend je Kategorie in einer Tabelle zusammen (S. 149, 176, 204).

*Exkurs: Technisch implementierte Sozialität –
Vertrauen und Kontrolle durch Open-PGP*

In den betrachteten Gemeinschaften geht es nicht zuletzt um gegenseitiges Vertrauen in die Interessen und Ziele von Menschen, da Software die DNA unserer täglichen „digitalen Assistenten“ darstellt und das Vertrauen in die Richtigkeit ihrer Funktionsweise hier ganz besonders an den Mitgliedern der Community hängt, die diese DNA verändern können und dürfen. Hilfreich für das Verständnis einiger unten beschriebener Vertrauensbeziehungen ist

44 Dabei beschränke ich mich vornehmlich auf die User- und Developer-Mailinglisten, also die zentralen Diskussionskanäle der Nutzungs- bzw. Entwicklungsperspektive.

45 In dieser Hinsicht ist meine Darstellung manchmal zugespitzt und vernachlässigt die individuellen Varianzen innerhalb der Communities zugunsten ihrer allgemeinen Tendenzen.

daher die technische Umsetzung von Vertrauen durch das kryptografische Verfahren OpenPGP⁴⁶, das ich vorneweg kurz erklären möchte.

Es handelt sich dabei um ein asymmetrisches Verschlüsselungsverfahren, bei dem jede*r Beteiligte einen „Schlüssel“ hat, der aus einer öffentlichen Komponente und einer geheimen Komponente besteht, also zwei Schlüsseldateien. Die geheime Schlüsseldatei liegt nur beim User, die öffentliche ist für gewöhnlich in öffentlich zugreifbaren Web-Datenbanken hinterlegt. Der öffentliche Schlüssel dient dazu, Dinge so zu verschlüsseln, dass sie nur mit dem passenden privaten Schlüssel wieder entschlüsselt werden können. So kann aber auch der Zugang bspw. zu einem Repository derart gestaltet werden, dass nur der oder die Inhaber*in des passenden privaten Schlüssels Zugriff erlangt. Mit dem privaten Schlüssel kann aber auch eine Signatur generiert werden, die mit dem passenden öffentlichen Schlüssel auf Gültigkeit geprüft werden kann. Wird eine Datei, ein Text oder ein öffentlicher Schlüssel signiert, kann mit dem passenden öffentlichen Schlüssel überprüft werden, ob der exakte Inhalt (die exakte Datei) unverändert vom Inhaber des privaten Schlüssels signiert wurde.

Da öffentliche Schlüssel – wie der Name schon sagt – nicht geheim sind, besteht hier ein gewisses Risiko, dass über einen gefälschten öffentlichen Schlüssel der oder die Inhaber*in eines dazu passenden anderen geheimen Schlüssels Zugriff erhält. Das Risiko wird verringert, indem andere Vertrauenspersonen den öffentlichen Schlüssel überprüfen und mit ihrem privaten Schlüssel signieren. Auf diese Weise werden Vertrauensketten in den betrachteten Communities aufgebaut. Der öffentliche Schlüssel eines neuen Mitglieds muss in manchen Zusammenhängen von anderen Mitgliedern signiert werden. Somit stehen diese signierenden Mitglieder Pate für die Korrektheit des Schlüssels und gewissermaßen auch für das Vertrauen, das dem oder der Inhaber*in des signierten Schlüssels entgegengebracht wird.⁴⁷

46 Es handelt sich um eine Weiterentwicklung von PGP, *Pretty Good Privacy*, als offener Standard. Gerne werden bei FLOSS-Reimplementationen die Buchstaben verdreht. Die FLOSS-Software, die den Standard OpenPGP nutzbar macht, heißt daher GPG, *GNU Privacy Guard*. Daher wird in der Praxis GPG, PGP und OpenPGP oft synonym verwendet.

47 Eine ausführlichere Beschreibung und weiterführende Links finden sich unter https://de.wikipedia.org/wiki/GNU_Privacy_Guard#Funktionsweise, letzter Aufruf 29.3.2018

8.1 Mitgliedschaft als kollektives Selbstbild

Zunächst ist Mitgliedschaft in freiwilligen Kollektiven häufig zu einem guten Teil ein Akt der nur bedingt beobachtbaren Selbstzuschreibung zum Kollektiv und damit schwer erfassbar. Dennoch gibt es explizite und implizite Kriterien, anhand derer sich die Mitgliedschaft zu einer Community näher bestimmen lässt. Zunächst werden von der Community Mitglieder nur wahrgenommen, wenn sie in irgendeiner Form in Erscheinung treten, beispielsweise ganz niederschwellig durch einen Diskussionsbeitrag in Form einer Frage. Ungeachtet dessen kann man ein gewisses Selbstbild der Community beobachten, das bestimmte Vorstellungen von Nutzer*innen impliziert, die im Umkehrschluss von der Community adressiert werden. Dies spiegelt sich wider in der Gewichtung der Prioritäten, beispielsweise in der Selektion und Konfiguration der Standard-Software oder der Abwägung zwischen Aktualität und Stabilität der Software, sowie in den Zielsetzungen der Community, beispielsweise der Gewinnung möglichst vieler neuer Nutzer (Ubuntu) oder der Bereitstellung eines möglichst individuell konfigurierbaren Systems (Arch). In dieser Hinsicht betrachte ich die Mitgliedschaft als eine Art kollektives Selbstbild der Community, das sich schließlich in impliziten und expliziten Voraussetzungen zeigt, die an Mitglieder gestellt werden, und entsprechenden Einstiegsangeboten für neue Mitglieder.

Dabei teilen sich die Voraussetzungen für die Mitgliedschaft auf in die Voraussetzungen für die Nutzung der Software einerseits und die Voraussetzungen für die aktive Teilnahme im Sinne eines Beitragens für die Community andererseits. Dies ist nicht vollständig zu unterscheiden, da es zwar die Möglichkeit der reinen Nutzung ohne aktive Partizipation gibt, aber die Nutzung des Systems immer potenzieller Anfang einer aktiven Community-Rolle ist und eine aktive Rolle in der Community in der Regel auch mit der Nutzung der Distribution verbunden ist. An dieser Stelle konzentriere ich mich auf die Voraussetzungen, die auch an die Nutzung gestellt werden, um mich im darauf folgenden Abschnitt (8.3) den konkreten Anforderungen an die spezifischen Beiträge zu fokussieren.

Schließlich gibt es auch formalisierte Formen der Mitgliedschaft, die teils verbunden sind mit institutionalisierten Bewerbungsverfahren. Diese sind als formalisierte Rollen im Grunde Teil der Strukturen und Verfahren, die in Abschnitt 8.2 beschrieben werden. Da sich aber in den unterschiedlichen Voraussetzungen und Bedingungen, die an die Mitgliedschaft gebunden sind,

das Selbstbild der Community widerspiegelt, werde ich in diesem Abschnitt die minimale formale Rolle, also die mit den geringsten Anforderungen, beschreiben.

8.1.1 Ubuntu – „Linux for Human Beings“

[S]aying RTFM⁴⁸ [...] is really not cool, nor following the [Code of Conduct].⁴⁹

Wie am proklamierten Ubuntu-Motto *Linux for Human Beings* deutlich wird, wird hier in erster Linie der „Mensch“ adressiert, und zwar als User von Technik, der sich zunächst als Laie die Technik zunutze machen möchte. Dementsprechend ist die Hürde des Einstiegs in die Community relativ niedrig gesteckt. Relativ ist hierbei ein wichtiges Wort, da für die Nutzung von GNU/Linux zunächst ein aktiver Schritt seitens der Users notwendig ist: Anders als bei den Betriebssystemen von Microsoft oder Apple sind GNU/Linux-Systeme in aller Regel nicht schon auf den im Handel erhältlichen Computern vorinstalliert, sondern müssen nachträglich installiert werden. Dies setzt einerseits die Kenntnis der Existenz von GNU/Linux als alternatives Betriebssystem voraus und andererseits die Bereitschaft zur Installation eines neuen Systems – eine deutliche Abweichung von der Norm. Das ist zwar nur ein geringes „Wagnis“, aber für einen Laien ist es nicht unbedingt abzusehen und einzuschätzen, was es bedeutet, die ominöse Betriebssystemebene zu verändern, was hierbei vielleicht schiefgehen könnte und wie dann zu verfahren wäre. Schließlich sind dadurch, dass die technische Ebene, solange sie ordentlich funktioniert, in ihrer Benutzung unsichtbar bleibt, auch technische Alternativen unsichtbar – und so muss ein Laie erst einmal erkennen, was ein Betriebssystem ist, um sich dessen bewusst zu werden, ein anderes installieren zu können.

Diese Überlegungen gelten zwar zunächst generell für alle Arten von GNU/Linux, treten aber in Bezug auf ein dezidiertes Einsteiger-Linux prominenter in Erscheinung, da hier Nutzer*innen adressiert werden, die bislang den vorgegebenen Standard nutzen – im Gegensatz zu den beiden anderen betrachteten Fällen, Debian Linux und Arch Linux, die nicht in erster Linie

48 RTFM steht für „Read the Fucking Manual“.

49 <https://lists.ubuntu.com/archives/ubuntu-users/2011-February/239679.html>, letzter Aufruf 5.5.2017

als Einsteiger-Systeme gedacht sind, wo also vielmehr Nutzer*innen adressiert werden, die schon GNU/Linux-Systeme kennen oder zumindest mit dem Konzept Betriebssystem vertraut sind.

Nutzerbild

Ubuntu adressiert durchaus Laien, die Computer vergleichsweise wie einen Fernseher benutzen, sozusagen also „Telly User“ (13,128) sind, also wenig Interesse an Hintergrundwissen und persönlicher Konfiguration haben. Dennoch ist das Sammeln von Erfahrung und Expertise im Umgang mit dem System durchaus möglich und auch für versiertere User lässt sich das System nach den gewünschten Bedingungen anpassen: “If you wanna be a system admin and run something – Ubuntu has that available, if you wanna just the user land – totally fine” (13,128).

Bemerkenswert ist, dass eine explizite Adressierung von Laien dazu führt, dass auch Laien sich in verschiedenen Bereichen aktiv mit Beiträgen beteiligen (11,127). Für Advocacy, Übersetzung und Support sind nicht unbedingt technische Kenntnisse der Funktionalität notwendig, sondern ein schematisches Wissen der Technik beziehungsweise Anwendungswissen (etwa *Beer-Mat Knowledge* und *Popular Understanding*, vgl. S. 76) reichen hierfür aus. Ein anwendungsbezogenes Verständnis der Software reicht teils sogar aus für einfache Bug Reports, insofern in Ubuntu auch weniger technische Bug Reports zulässig sind. Ausführlicher werden die verschiedenen Beitragsmöglichkeiten und die damit verbundenen Voraussetzungen in Abschnitt 8.3.1 beschrieben.

Laien werden also als reine Nutzer*innen akzeptiert, die das System lediglich *benutzen* wollen, ferner wird ihre reine Nutzerperspektive in den Beiträgen angenommen. Das Nutzerbild adressiert also Laien in der Nutzungs- wie auch in der Feedback-Rolle als Beitragende.

Voraussetzungen

Ein Interviewter nennt eine „Do-It-Yourself-Mentalität“ (6,79) als Voraussetzung für eine beitragende User-Rolle und zu einem gewissen Grad auch als allgemeine Prämisse, um sich überhaupt darauf einzulassen, ein alternatives Betriebssystem zu installieren.

“To actually get involved in a project? Then yeah, they probably have to have a bit of a help-yourself-mentality. I mean they have to – googling to try things. But, you know if not, they wouldn’t be users in the first place.” (6,79)

Die notwendig vorgelagerte Entscheidung für GNU/Linux als alternatives Betriebssystem kann als grundsätzliche Voraussetzung für den Einstieg in eine Linux-Community betrachtet werden. Als Motivation für den Systemwechsel nennt ein User *Leidensdruck* (9,208), in seinem Falle Unzufriedenheit mit dem vorinstallierten Betriebssystem MS Windows.

Die Voraussetzungen für eine Teilnahme an der Community spielen eine tragende Rolle für die Eintrittsmöglichkeiten in die Community und spiegeln das Selbstbild ihrer Mitglieder wider. Wenngleich es auch bei Ubuntu sicherlich äußerst hilfreich ist, ein gewisses Verständnis von Grundlagen der Betriebssysteme zu haben – wie zum Beispiel, was eine Partitionierung von Festplatten bedeutet (9,21) –, so kommt man hier auch ohne dieses Wissen deutlich leichter zu einer erfolgreichen Installation als in den anderen beiden betrachteten Fällen. Gerade in diesem Aspekt weisen die Fälle eine interessante Varianz auf, da grundlegendes Wissen hier (Ubuntu) möglichst einfach erklärt und dort vorausgesetzt⁵⁰ wird (Arch). Darüber hinaus ist diese Voraussetzung (oder auch Nicht-Voraussetzung) von Wissen in das Installationsprogramm eingeschrieben, wie ich weiter unten darstellen werde (s. Kap. 9).

Bei einer auf Einsteiger*innen ausgerichteten Community wie Ubuntu spielt der Einstieg in die *Nutzung* eine besondere Rolle, während, wie ich unten ausführe, in einer Distribution, die auf Entwickler*innen fokussiert, der Einstieg viel stärker auch ein Einstieg in die *aktive Gestaltung* durch eigene (Code-)Beiträge bedeutet. Dies geht wiederum einher mit unterschiedlichen Voraussetzungen an die Mitgliedschaft. Es wird deutlich, dass die unterschiedlichen Dimensionen der Variable stark zusammenhängen.

Die Einordnung von Beiträgen hängt wiederum ab vom Selbstbild der Community: Wird die eingeforderte Hilfestellung (Support-Anfrage) und Beteiligung schon als Teil der *aktiven* Teilnahme an der Community gewertet? Denn in der Interaktion von Frage und Antwort, in Form einer öffentlichen Kommunikation, ergibt sich aus der Support-Anfrage durch die dokumentierte Befolgung der Ratschläge und der Mitteilung der Zwischenergebnisse eine Problembeschreibung mit Lösungsmöglichkeiten und dient idealiter zukünftigen Usern wiederum als Wissensquelle. Somit kann das Stellen selbst einfacher Fragen als Beitrag gelten und ist im hier betrachteten Falle schon der Einstieg in die aktive User-Rolle.

50 im Sinne eines eigenen Erarbeitens des notwendigen Wissens im Geiste von RTFM

In der praktizierten Offenheit und der Akzeptanz selbst einfacher Fragen spiegelt sich also das Selbstverständnis der Community wider: „Die Community da ist sehr aktiv und ist sehr darum bemüht, halt auch wenn du dann schreibst, du bist kein Experte, die geben sich Mühe und schreiben so, dass man’s versteht – als normaler Mensch“ (9,66).

Eine grundlegende Rolle spielen dabei das Bewusstsein, selbst einmal Einsteiger gewesen zu sein, sowie die gemeinsame Übereinkunft, möglichst freundlich und hilfsbereit zu sein. Dies ist explizit in einem *Code of Conduct* festgeschrieben, der einen maßgeblichen Einfluss auf die Konstitution der Community ausübt (3,361) und den Willen zur Inklusion von Laien fest schreibt. Verstöße gegen den *Code of Conduct* werden durchaus sanktioniert, beispielsweise können unflätige Kommentare in Internet Relay Chats (IRC) dazu führen, dass der oder die Delinquent*in aus den Chat „gekickt“ wird (6,185).

Die Fragen der Einsteiger*innen prägen wiederum einen einstiegfreundlichen Wissensfundus, da in der dokumentierten Kommunikation auch die Fragen von Neulingen mitsamt passenden Lösungsvorschlägen auftauchen, Fragen, auf die erfahrenere User mitunter gar nicht kommen (13,48). Als hilfreich beim Einstieg beschreibt ein User das Vorhandensein einer ansprechenden, informativen Hilfeseite (ubuntusers.de)⁵¹, wo leicht zu verstehende Anleitungen zu finden sind, die aber dennoch den „nötigen Tiefgang“ bieten (9,21). Die Einsteigerperspektive findet sich also nicht nur in der direkten Kommunikation wieder, sondern geht offensichtlich ein in die Informations- und Hilfeseiten der Community.

Als Voraussetzung für die Beteiligung in der Community reicht also im Grunde ein schematisches Wissen über das System (*Beer Mat Knowledge*, vgl. Abschnitt 5.1) zunächst aus. Der User kann relativ niederschwellig in der Interaktion mit der Community und durch die Auseinandersetzung mit der vorliegenden Dokumentation weitergehendes Wissen erlangen.

Einstieg

Bezüglich der Fähigkeiten, die von Usern erwartet werden, ist bislang deutlich geworden, dass hier relativ wenig Wissen vorausgesetzt wird und die notwendigen Informationen möglichst niederschwellig angeboten werden. In

⁵¹ in diesem Falle eine Hilfeseite für den deutschsprachigen Bereich, die aber aufgrund ihrer reichhaltigen Beschreibungen auch von anderen Linux-Communities empfohlen wird

dieser Hinsicht betrachten auch versierte Nutzer, die gewisse Erfahrungen mit unterschiedlichen GNU/Linux-Systemen haben, Ubuntu als eine geeignete Einsteigeroption, die sie beispielsweise ihren Familienangehörigen (10,50) oder anderen *End-Usern* empfehlen: “I also run Ubuntu, because I want an option, I can give people a try” (13,14).

Der Einstieg in die Nutzung eines GNU/Linux-Systems ist implizit stets verbunden mit der *potenziellen* aktiven Beteiligung an der Community und darüber hinaus der Sammlung von Erfahrungen und Weiterentwicklung der eigenen Kenntnisse und gegebenenfalls dem Umstieg zu einer anderen Linux-Community. Insofern zeigt sich hier *Einstieg* ebenfalls mehrdeutig als Einstieg in die Nutzung von Ubuntu, die Partizipation in der Community und weiter als Einstieg in die GNU/Linux-Welt. Insofern erfüllt die Ubuntu Community auch eine Funktion für die gesamte FLOSS-Community, indem hier in besonderem Maße neue User („Umsteiger*innen“) mit GNU/Linux sozialisiert werden.

Der Einstieg in die Community erfolgt auch über lokale Anwendertreffen und Nutzergruppen, wo zunächst Überzeugungsarbeit für das Produkt geleistet wird („evangelize“) und Erfahrungen im Umgang mit dem System ausgetauscht werden. Hier können auch die notwendigen Skills ausgetauscht und entwickelt werden, um in weiteren Bereichen Beitragsmöglichkeiten zu finden.

Vor allem in den ersten Jahren (2005–2008) nach der Gründung von Ubuntu gab es in Deutschland eine Vielzahl von Ubuntu-Stammtischen und Release-Partys, auf denen gemeinsam die neuesten Releases installiert wurden und User sich gegenseitige Hilfestellung gaben. Dies ist aber seit 2008 deutlich weniger geworden und die Ubuntu Community hat sich in die (schon vorher) bestehenden Strukturen von Linux-User-Gruppen und Linux-Install-Partys eingefügt.⁵² Die breitflächige Entstehung von Ubuntu-Stammtischen in der Anfangsphase verdeutlicht die Aufmerksamkeit, die Ubuntu in den ersten Jahren erzeugt hat. Im Laufe der Zeit haben sich die neuen User vielerorts in die breite Community der GNU/Linux User integriert und in die langjährigen Strukturen von Linux User Groups. Lokale Anwendertreffen sind im Grunde nichts Ubuntu-spezifisches, sondern haben eine längere Tradition. Bemerkenswerterweise wurde in den Interviews mit Mitgliedern der

52 Vgl. Vortrag von Torsten Franz auf dem Chemnitzer Linux-Tag 2016 (insbesondere ab Minute 22), <https://chemnitzer.linux-tage.de/2016/de/programm/beitrag/209>, letzter Aufruf 15.3.2017.

Ubuntu Community mehrfach auf derartige Treffen verwiesen. Dies steht im Gegensatz zu den anderen Interviews, wo derartige Treffen nicht als Form von aktiver Beteiligung Erwähnung finden.

Wenn auch manche User ihre persönlichen Fähigkeiten als zu gering erachten, um an lokalen Treffen teilzunehmen (9,125f), werden sogenannte *Meet-ups* explizit als niederschwelliger Einstieg in eine aktive Rolle der User innerhalb der Community betrachtet (mehr dazu siehe 8.3.1):

... So you're an Ubuntu user, and in one time you go to some meet-up ... and so the next stage is to get involved in those local communities and start advocating, go to conferences like this one ... that's like – you only have to be a user ... that's really easy, to bridge the barrier. Then afterwards it's contributing directly to the production of the software, ... and then you have to leverage some body of skills. (4,148ff)

Der Einstieg als Laie in eine aktive Rolle in der Community wird von der Community als sinnvolle Option betrachtet und auch das virale Marketing als Dienst für die Gemeinschaft wertgeschätzt, der von Usern gerade auch als Laie erbracht werden kann. Zudem wird eine Lernerfahrung in der Community impliziert, die weitere Möglichkeiten der aktiven Beteiligung eröffnet. Unabhängig aber vom Expertisegrad eines beitragswilligen Users werden verschiedenste Fähigkeiten, insbesondere auch solche, die nicht direkt mit dem technischen Charakter des Wissensprodukts zusammenhängen, gewürdigt. Dies impliziert auch eine Toleranz gegenüber Anfängerfehlern und einer variierenden Qualität der Beiträge: “We try not to discriminate, [...] we won't dismiss people that don't necessary good at [writing bug reports] – if there is a bug file, if there is enough details to reproduce, we won't say ‘ok this bug is written wrong’” (5,57f).

Formale Mitgliedschaft

Bemerkenswert ist, dass es im Fall von Ubuntu – im Gegensatz zu den anderen Fällen – eine Form der Mitgliedschaft gibt, die völlig unabhängig von der Tätigkeit der Software-Entwicklung ist.

Für eine volle Mitgliedschaft mit Mitbestimmungsrecht bei Wahlverfahren (sogenannte *Resolutions* und Wahlen zur Besetzung des *Community Council*) bedarf es eines formalen Bewerbungsverfahrens, in dem erbrachte Beiträge für die Community benannt werden müssen. Dies schließt aber ex-

plizit Beiträge ein, die unabhängig von der technischen Entwicklung des Systems sind (3,294ff).⁵³ Auf der Wiki-Seite heißt es im Wortlaut:

The idea that only technical contributions (like patches or uploads of packages) or only contributions which give rise to karma in Launchpad is incorrect. The community is much broader and more diverse than that and it is sufficient to demonstrate significant and sustained contributions in any area of the Ubuntu community.⁵⁴

Bewerber*innen benötigen für das Verfahren Fürsprecher*innen, die die Aktivitäten der Bewerber*innen bezeugen und damit die Bewerbung unterstützen. Im *Membership Board* wird dann über den Mitgliedsantrag entschieden. Wer anerkanntes Mitglied wird, erhält auch eine E-Mail-Adresse bei ubuntu.com und eine vom Ubuntu-Gründer und Canonical-Chef unterzeichnete Mitgliedschaftsurkunde. Diese Respektbekundung stärkt den Community-Charakter – und die Ubuntu-Mailadresse ist geeignet, die Verdienste des anerkannten Mitglieds auch nach außen kenntlich zu machen und verleiht somit dem *Ubuntu Member* eine gewisse Autorität.

Niederschwelliger und lediglich gebunden an die Signierung des *Code of Conduct* mittels eines kryptografischen Schlüssels – als sichtbares Commitment zu den Werten der Community⁵⁵ –, kann jede*r „Ubuntero“ werden. Dies erfolgt über einen Account auf der *Launchpad*-Plattform, die als zentrales Werkzeug der Koordination zwischen den Mitgliedern dient, und auf der jedes Mitglied mit seinen Community-Aktivitäten sichtbar werden kann (vgl. Hill u. a. 2006: 305), und hat somit eine identitätsstiftende Funktion. Diese Form bringt jedoch keine formalen Rechte in der Community mit sich und wird mangels eines Aufnahmeverfahrens auch nicht durch andere Mitglieder der Community bewertet – und findet in den Interviews auch keine explizite Erwähnung.

53 Es sei an dieser Stelle angemerkt, dass es inzwischen auch bei Debian vereinzelt Mitglieder gibt, die zwar den Status eines *Debian Developer* haben, aber keine Schreibrechte am Repository. Dies verweist auf einen Mitgliedsstatus ohne Entwickler-tätigkeit, der aber bislang eher die Ausnahme als eine reguläre Praxis darstellt und weiterhin unter der Bezeichnung „Developer“ geführt wird.

54 <https://wiki.ubuntu.com/Membership>, letzter Abruf 15.3.2017

55 Das auf S. 119 beschriebene Verfahren wird hier verwendet, um mit dem individuellen privaten Schlüssel den *Code of Conduct* technisch überprüfbar zu unterschreiben.

8.1.2 Debian – „The Universal Operating System“

*When the code is public, RTFM is the proper answer.
One might add: “document it properly afterwards”.*⁵⁶

Debian formuliert mit dem Motto eines universellen Betriebssystems⁵⁷ einen hohen Anspruch, der sich widerspiegelt in einer überdurchschnittlich hohen Anzahl an unterstützten Rechnerarchitekturen sowie bereitgestellten Softwarepaketen.⁵⁸

Nutzerbild

Wenngleich die Berücksichtigung von Anfänger*innen auch in Debian zunehmend eine Rolle spielt – etwa durch eine vereinfachte Verwendung proprietärer Treiber in der Installation oder Empfehlungen für Anfänger*innen im Installationsprogramm (vgl. Abschnitt 9.2) –, wurde es Ende der 1990er bekannt als ein System für “serious, technically-capable Linux users”⁵⁹ und wird auch heute entsprechend gerne als schlankes, robustes System für den Serverbetrieb verwendet (2,181f).

Die Voraussetzungen, die Nutzer*innen und Einsteiger*innen mitbringen sollen oder die implizit erwartet werden, umfassen ein hohes Maß von Bereitschaft, sich mit dem System zu beschäftigen: Benötigte Dokumentationen zu finden ist nicht immer ganz einfach (2,103) und wie im Eingangszitat des Abschnitts deutlich wird, gilt die Prämisse, dass gar die notwendigen Informationen aus dem Softwarecode abgeleitet werden können – die Fähigkeit, diesen zu verstehen, wird hierbei implizit vorausgesetzt (vgl. auch Abschnitt

56 <http://www.mail-archive.com/debian-vote@lists.debian.org/msg08500.html>, letzter Aufruf 3.1.2017, auch zitiert in Coleman 2013: 111

57 Siehe die Website des Projekts <https://www.debian.org/>.

58 Da es unterschiedliche Prozessoren-Typen gibt, die über unterschiedliche Befehle angesprochen werden, ist es für jeden Typ von Prozessor notwendig, den Quellcode entsprechend des Typs zu kompilieren. Debian bietet derzeit 43.000 fertig kompilierte Programmpakete, das aktuelle „stable Release Stretch“ unterstützt zehn Architekturen: x86 (amd64 & i386), ARM (arm64, armel, armhf), MIPS (mips, mips64el, mipsel), PowerPC (ppc64el), s390x; Quelle: <https://de.wikipedia.org/wiki/Debian#Architekturen>, letzter Aufruf: 8.3.2018.

59 <https://www.debian.org/doc/manuals/project-history/ch-detailed.html>, letzter Aufruf 15.5.2017

8.1.2). Hier spiegelt sich die Adressierung von ernsthaften, technisch versierten Nutzer*innen wider.

Eine Orientierung an den Bedürfnissen eher professioneller Anwender*innen zeigt sich auch in den hohen Qualitätsanforderungen und der Fokussierung auf Universalität und Stabilität, was vor allem im Server-Bereich gefragt ist. Die Universalität beispielsweise im Sinne einer großen Bandbreite verschiedener unterstützter Rechnerarchitekturen ist für normale Endanwender („Desktop-User“) insofern weniger wichtig, als in diesem Feld nur wenige etablierte Rechner-Architekturen zum Einsatz kommen. Diese Eigenschaft zielt also auf „Power-User“, Administratoren oder Bastler, die verschiedenste Computer betreiben.

Eine besondere Fokussierung auf Stabilität zeigt sich in der Organisation der unterschiedlichen Varianten oder *Distributions-Kanäle*. Wie auch bei anderen FLOSS-Projekten üblich, kann man wählen zwischen einer aktuelleren Version, die „mit heißer Nadel gestrickt“ ist und daher noch Fehler enthalten kann, und einer stabilen Version, die noch nicht die neuesten Features beinhaltet, aber dafür eine höhere Stabilität aufweist. Bei Debian fließen die aktuellen Änderungen über die Kanäle *experimental* und *unstable* nach einer Bewährungsfrist zunächst in das sogenannte *testing*. Dieses wird ca. alle zwei Jahre, aber immer dann, wenn „es eben soweit ist“, in ein *stable* Release überführt. Dem geht eine Periode des *freeze* voraus, in der keine neuen Features eingearbeitet werden, damit alle Konzentration auf die Behebung der bekannten Fehler gelenkt werden kann. Neben der neuen *stable* Version wird die Vorversion aus Gründen der Kontinuität und Kompatibilität noch als *old-stable* weitergeführt. Die *stable* Varianten bekommen über die Zeit noch Sicherheitspatches, das heißt, sicherheitskritische Änderungen werden in den Code eingepflegt, während neue Funktionen, die wiederum neue Fehler oder Inkompatibilitätsprobleme mit sich bringen könnten, außen vor bleiben. Durch diese Art der Differenzierung ist *stable* ein wirklich stabiles System, während aber durch die verschiedenen Kanäle wiederum eine Universalität im Sinne verschiedener Reifegrade von „stabil“ bis „bleeding edge“ (auf dem allerneuesten Stand) erreicht wird.

Betreiber*innen von Servern haben deutlich höhere Anforderungen an Stabilität und Kontinuität als Desktop-User, da der Absturz eines Desktop-Rechners eher zu verkraften ist als der eines Servers, insofern als nur ein User betroffen ist, während bei einem Server in der Regel viele User betroffen sind und diese häufig Funktionalitäten zur Verfügung stellen, die für Organisationen eine zentrale Bedeutung haben. Hingegen sind neue Features

eher für Desktop-User (tendenziell Endanwender*innen) wichtiger, beispielsweise zur Nutzung neuer Funktionen in Büroanwendungen. Allerdings wollen Desktop-User nicht unbedingt die unausgereiften fehlerhaften Entwicklerversionen nutzen. Schließlich haben Entwickler*innen in der Regel ein Interesse daran, Zugriff auf die neuesten Versionen zu haben, da dies die Versionen sind, an denen auch aktuell entwickelt wird, beziehungsweise greifen Entwicklerversionen zurück auf entsprechend aktuelle Versionen anderer Programme und Bibliotheken und benötigen diese, um ordentlich zu funktionieren.

Kontinuität und Stabilität stehen mit der Variante *stable* und der erweiterten Pflege der überholten *stable*-Versionen in *old-stable* klar im Vordergrund. Dies zeigt sich an der grundsätzlichen Empfehlung, *stable* zu nutzen, sowie der Policy, dass *testing* zu *stable* wird, „wenn es eben fertig ist“ und keinesfalls zu einem verfrühten Zeitpunkt. Hier ist keine Eile zu erkennen, neue Features verfügbar zu machen – wichtiger ist, dass die Systeme ordentlich weiterlaufen. Dies kritisiert ein User, der meint, *stable* sei im Grunde veraltet, und *testing* sei hinreichend stabil (13,33) – aber er spricht eben aus der Perspektive eines Endanwenders (Desktop-User) und illustriert in seiner Aussage die Wirkung der User-Adressierung der Community:

In Debian [...] *because they have the wrong terms, the entire kind of focus is use stable. When a new person comes to the distribution, [...] stable on debian is really old, like it's everything is outdated. But that's part of the point, it's a solid system, doesn't change often, except for security patches, so it's stable. But as far as a home user that is not something you want. (13,33)*

Die Kritik zielt an dieser Stelle darauf, dass viele User enttäuscht sind von Debian, weil auf der Website des Projektes empfohlen wird, *stable* zu nutzen, was aber für Desktop-User eine schlechte Wahl sei und eher für professionelle „User“ und Administrator*innen Sinn ergibt. Umgekehrt kann man daraus ableiten, dass diese Desktop-User schlicht nicht dem anvisierten Nutzerbild entsprechen, sondern hier eher Administrator*innen und Betreiber*innen von Servern beziehungsweise Power-User adressiert werden, die ein starkes Bedürfnis nach Stabilität haben oder gut genug Bescheid wissen, um entgegen der Empfehlung ein *testing*- oder *unstable*-System zu betreiben.

Die Fokussierung auf Freie-Software-Prinzipien in den Policies des Projektes ist einerseits für viele Power-User wichtig, die Software so auf verschiedene Arten weiterverwenden oder verändern zu dürfen, andererseits zeigt sich in der Priorisierung des Prinzips Freie Software eine politische Dimension, die das Prinzip Freie Software als Leitbild definiert. Auch die

starken demokratischen Elemente in der Entscheidungsfindung haben eine politische Dimension. Reine Nutzer*innen haben hierin jedoch keine Funktion oder Stimme, sondern nur formal anerkannte *Debian Developer* haben Stimmrechte. Power-User, die sporadisch beitragen, aber keine formale Rolle innehaben, sind hierbei ebenfalls wenig repräsentiert. Die Strukturen der Organisationen tragen also politische Züge, am meisten profitieren davon aber aktive Entwickler*innen.

Zusammenfassend kann man sagen: Das beobachtete Nutzerbild adressiert besonders die erweiterte und professionelle Nutzung des Systems und legt Wert auf Prinzipien wie Freie Software und strukturelle Transparenz (vgl. Abschnitt 8.2.2).

Voraussetzungen

Auf die Frage nach den Voraussetzungen für die Nutzung von Debian antwortet ein Interviewter lakonisch: “You have to know that Debian *exists*” (2,177). Dies ist zwar eine allgemeingültige Feststellung, die gewissermaßen auf jedes System oder Software-Projekt zutrifft, gibt aber dennoch einen Hinweis auf die Voraussetzungen, die man als User hier mitbringen muss: Debian ist zwar innerhalb der Linux-Welt ein wichtiger Bezugspunkt, ist aber nicht unbedingt die Distribution, die unbedarfte Linux-Einsteiger*innen kennen. Eine wesentliche Bedeutung kommt Debian schließlich zu, da unzählige andere Distributionen darauf aufbauen (wie auch der hier betrachtete Fall Ubuntu).

Weiter wird ein gewisses Interesse an der Funktionsweise des Systems nahegelegt, da dies zielführend und mitunter auch notwendig für die Konfiguration ist. (2,179ff; 13,128). “Anyone *can* use it, but you kind of want to know what your system is – you wanna kind of know the things are happening behind your back I guess ...” (13,128). Folglich liegt es nahe, dass User bestimmte Gründe haben, dieses System zu benutzen, beispielsweise weil für den Server-Betrieb hohe Anforderungen an Stabilität und Konfigurierbarkeit bestehen und sie dafür keine grafischen Konfigurations-Werkzeuge benötigen (2,181f).

Für die erfolgreiche Installation und Konfiguration von Hardware ist mitunter auch ein erweitertes Verständnis notwendig, welche Hardwaretreiber proprietär sind und wie man diese bekommt. Konkret sind in Debian eine Menge proprietärer Treiber zunächst nicht auf dem Installationsmedium enthalten, da sie gegen die Anforderungen der *Debian Free Software Guidelines*

verstoßen.⁶⁰ Dies führt dazu, dass manche Hardware zunächst nicht funktioniert. Besonders schwerwiegend ist dies bei Hardware, die den Zugang zum Internet ermöglicht, da deren Nicht-Funktionieren den Zugriff auf die notwendige Dokumentation und die entsprechenden Treiber verhindert, die üblicherweise im Internet zu finden sind.

Web-Entwickler: You have to have a friend and help try to install everything, ((Debian-Entwickler lacht herzlich)) you have to have a *friend* (.) who knows Debian because then when you install it and your Wifi doesn't work, and like nothing, you can ask him.

Debian-Entwickler: That's true. (2,179)

Diese Eigenheit liegt in einer besonderen Verpflichtung gegenüber den Leitsätzen Freier Software und dem Ziel, "to provide an integrated system of high-quality materials with no legal restrictions", begründet. Diese sind im *Social Contract*⁶¹, einem zentralen Dokument, das die Leitsätze der Community formuliert, festgeschrieben. In dieser Hinsicht ist schon die Bereitstellung proprietärer Software ein Schritt zur Unterstützung der User, die auf derartige Anwendungen angewiesen sind, auch wenn dafür manuell die entsprechende Quelle (Repository) eingerichtet werden muss.⁶²

Es zeigt sich hier, dass die Nutzung von Debian in mancher Hinsicht voraussetzungsvoller ist als Ubuntu, wo die Installation proprietärer Treiber sehr niederschwellig und ohne Umschweife möglich ist. Für eine erfolgreiche Nutzung bedarf es daher gewisser Kenntnisse bezüglich der Funktionsweise des Computers – etwa der Notwendigkeit und Beschaffenheit von Treibern – oder zumindest die Bereitschaft, sich damit auseinanderzusetzen und sich diese Kenntnisse anzueignen. Dies ist zunächst nicht unbedingt eine gewollte Prämisse, sondern ergibt sich im genannten Beispiel der WLAN-Karte aus den Besonderheiten der Hardware (proprietäre Treiber sind erforderlich) und den politischen Leitlinien der Community. Dies kann einen Laien überfordern, insofern dieser nicht versteht, wo und warum das Problem auftritt und wie es zu lösen wäre. Die normativen Setzungen, die sich in den verschrift-

60 Genauer gesagt müssen die non-free repositories an geeigneter Stelle ergänzt werden, damit die dort befindlichen Pakete zur Installation zur Verfügung stehen, oder sie müssen separat gefunden und eingebunden werden (13,49).

61 Siehe https://www.debian.org/social_contract.

62 Diese Repositories waren Gegenstand kontroverser Diskussionen über die Wahlfreiheit, auch proprietäre Software zu nutzen, gegenüber der Intention, proprietäre Software politisch nicht zu unterstützen (vgl. dazu Lazaro 2009).

lichten Regeln widerspiegeln, wirken also zurück auf das Wissen, das vom User vorausgesetzt wird. Somit erfahren potenzielle User einen Lernprozess, in deren Verlauf sie sich Wissen aneignen und Verständnis entwickeln – oder sie wenden sich ab und werden nicht Teil der Community. Die Voraussetzungen der Nutzung des Systems wirken somit auch als Selektor für die aktive Partizipation an der Community, da User sich gewisses Wissen aneignen müssen, um erfolgreiche User zu sein.

Für eine aktive Rolle in der Community haben die Policies der Debian Community eine wichtige Bedeutung, wie sich bei der Kontribution und den daran geknüpften Erwartungen zeigt: Wie im Abschnitt 8.3.2 deutlich wird, sind für die erfolgreiche Mitarbeit gewisse Regeln zu beachten (4,90f), beispielsweise, in welcher Form Bugs dokumentiert werden und auf welche Weise Software-Pakete gebaut werden. Der Verweis auf gewisse Standards bei Beiträgen ist durchaus üblich in derartigen kollaborativen Projekten, aber Debian-Entwickler selbst räumen ein, dass diese Policies teilweise recht rigide durchgesetzt werden und somit auf manche abschreckend wirken. Ein Entwickler, der in Ubuntu und Debian beiträgt, stellt fest: “I know people who are afraid to go on debian-devel”⁶³ (6,172).

Ein anderer Entwickler, der sowohl in Debian als auch Ubuntu beiträgt, bezeichnet die Debian Community recht drastisch als eine „elitist group“, in der die Prinzipien wichtiger sind, als neue User und Entwickler zu gewinnen:

... they are not trying to sell a product, they are not really trying to get more users, what’s important is the rules, the policies, the respect of the policies on there. They will be perfectly happy to lose the last non-debian developer user, if that’s what it takes to keep purity (4,88).

Hier offenbart sich eine gewisse Spannung zwischen den beiden Communities, deren tiefere Erörterung mir der vorhandene Platz nicht gestattet. Es wird hier aber deutlich, dass es unterschiedliche Bewertungen von Integration von Usern und Entwickler*innen gibt und unterschiedliche Sichtweisen auf Software. Software wird bei Ubuntu eher auch als ein Technikprodukt betrachtet, also als eine „Sachtechnik“, die von Usern leicht benutzt werden kann. Bei Debian wird Software hingegen eher als ein Wissensgegenstand betrachtet, den man sich aneignen muss und bei dessen Weiterentwicklung es wichtig ist, bestimmte (tradierte) Standards und Prinzipien einzuhalten.

63 „Devel“ bezeichnet die Mailingliste für die Belange der Debian-Entwicklung, <https://lists.debian.org/debian-devel/>, letzter Aufruf 17.3.2017.

Positiv gewendet nennt ein Debian-Entwickler als Voraussetzungen für eine aktive Beteiligung Freund*innen, die schon im Projekt involviert sind und unterstützend zur Seite stehen bei der Suche von Dokumentation, beim Einhalten der Regeln, und die als Mentor*innen (s. Abschnitt 8.1.2) beim Einstieg in die Community helfen (2,111f). Die Mentorenschaft ist fester Teil des Bewerbungsverfahrens für eine formale Mitgliedschaft, die in Debian besonders bürokratisch ausgeprägt ist. Dies hängt schließlich zusammen mit dem unglaublichen Wachstum der Debian-Entwickler-Community und ihrem Anspruch, eine egalitäre Governance-Struktur zu pflegen und ein hohes Level an technischer Qualität und normativer Integrität zu gewährleisten.⁶⁴ Aus dem normativen Anspruch ergeben sich teils sehr politische Diskussionen innerhalb der Mailinglisten – “too political, lengthy hairy discussions” (11,47) –, nicht zuletzt eine Konsequenz der egalitären Entscheidungsstrukturen innerhalb der Debian Governance, dies wird näher diskutiert in Abschnitt 8.2.2.

Da in Debian ein außerordentlich stark formalisiertes Bewerbungsverfahren existiert, enthält dies auch sehr explizite Vorgaben, was von Mitgliedern erwartet wird, mehr dazu in Abschnitt 8.1.2. Anders als bei Ubuntu bezieht sich der formale Mitgliedsstatus primär auf Entwickler*innen, wie sich schon unschwer an der Bezeichnung *Debian Developer* erkennen lässt; in dieser Hinsicht verbirgt sich die Fähigkeit, mit Softwarecode umzugehen, hier implizit als Voraussetzung für eine aktive Rolle in der Community.

Insgesamt betrachtet ist für die Nutzung des Systems eine gewisse Auseinandersetzung mit den technischen Zusammenhängen hilfreich. Das nötige Vorwissen kann sich der geneigte User aber beispielsweise zunächst über einschlägige Computerzeitschriften aneignen. Als minimale Voraussetzung sollte hier also *Popular Understanding* ausreichen (vgl. Abschnitt 5.1).

Einstieg

In den Interviews wurde bemerkt, dass der Einstieg in die Community nicht immer leicht fällt, was im Zusammenhang mit starker Regelerorientierung, einer inneren Geschlossenheit der Entwicklergemeinschaft, aber auch mit der Beschaffenheit der technischen Werkzeuge der Zusammenarbeit steht; darüber hinaus wurden auch Defizite an Einstiegsangeboten bzw. im Mentoring thematisiert.

⁶⁴ Dies wird beispielsweise deutlich an den genannten *Debian Free Software Guidelines* ausführlich zur ethischen Fundierung von Debian vgl. Coleman (2013).

Wie im vorigen Abschnitt zur Sprache kam, ist es besonders hilfreich, jemanden zu haben, der einem die Türen öffnet und unterstützt. Ist dies offenbar schon für den Einstieg in die Nutzung wichtig (2,179), so gilt dies viel mehr noch für den Einstieg in die Kontribution (2,111f). Aufgrund der hohen Bedeutung der Einhaltung von Policies und Regeln und dem Teils sehr direkten Umgang damit erweist sich der Einstieg in die aktive Community als schwierig (4,62). Policies sind nicht nur für die Gestaltung von Software-Paketen zu beachten, sondern auch für das korrekte Bug Reporting über die Mailingliste (s. auch S. 198). Somit stellt auch die technische Organisation der Kommunikation für manche eine Einstiegshürde dar: “Debian is tricky, Debian is a tricky project, cause everything is happening on the mailinglist, which is quite hard at the beginning, and some people can be rude, and Debian – it is hard to start contributing to Debian” (2,84). Über die Mailingliste bekommen Neulinge gerne zum Einstieg ein Software-Paket zugewiesen, das „verwaist“ ist, also aktuell von niemandem betreut bzw. verwaltet wird. Dies wirkt offensichtlich nicht immer unbedingt attraktiv: “... for Debian it’s like get on the mailing list and have someone hand you a deprecated package, that *sounds sooo exciting* ((lacht)) (2,92)”. Einstieg heißt hierbei offenbar tendenziell Einstieg durch technische Beiträge, anders als dies beispielsweise bei Ubuntu der Fall ist.

Wenngleich auch, wie oben (Abschnitt 8.1.2) beschrieben, keine aktive Werbung neuer Mitglieder betrieben wird (4,173), so gibt es offenbar doch auch ein Problembewusstsein, dass aufgrund mangelnden Mentorings Potenziale verloren gehen. Beispielsweise gab es Versuche seitens des Debian Project Leaders, eine Art Mentoring-Programm voranzubringen (4,92,98–100). Ein Art Problembewusstsein findet sich auch in ironischer Selbstkritik auf der Mentoren-Frageseite:

What can I do to help Debian other than packaging? Excellent question! Debian is a lot more than a great big collection of random packages. There’s documentation to write and translate, bugs to fix, installers to test, and newbies to harass[...] uh, help.⁶⁵

65 https://wiki.debian.org/DebianMentorsFaq#What_can_I_do_to_help_Debian_other_than_packaging.3F, letzter Abruf 15.3.2017

Dementsprechend gibt es auch eine Debian-Mentoring-Liste, die auch niederschwellige Fragen zum Thema Kontribution ermöglichen soll: “[W]e aim to be the ‘softer, gentler’ Debian development forum”⁶⁶.

Teil der Bemühungen, den Einstieg in die Kontribution zu vereinfachen, ist auch die Klassifizierung von „Newcomer“-Bugs – die zumindest aus Entwickler-Perspektive relativ einfach sind (2,94ff) – und ein Software-Programm mit dem programmatischen Namen „how-can-i-help“ (12,7), das nach jedem Update und nach jeder Software-Installation eine Menge von zu erledigenden Aufgaben auflistet, basierend auf den jeweiligen Programm-Paketen, die auf dem betreffenden System installiert sind. Den Newcomer Bugs attestiert ein Interviewter aber mäßigen Erfolg und eine Liste der zu offenen Aufgaben erleichtert selbst noch nicht den Einstieg in die Strukturen und Praktiken, diese auch zu erledigen (mehr dazu in Abschnitt 8.3.2).

Beim Einstieg in die Community sind einige Regeln und Gepflogenheiten der Community zu beachten. Dies spiegelt sich auch wider im Bewerbungsverfahren zur formalen Mitgliedschaft, das im nächsten Abschnitt beschrieben wird. Positiv gewendet kann dies auch als umfassende Sozialisierung neuer Mitglieder betrachtet werden. Das Bewerbungsverfahren baut im Grunde auf wiederholtem Beitragen auf, das zunächst nur über Mitglieder der Developer-Community erfolgen kann. Diese können darauf hinweisen, wo aktuelle Dokumentation zu finden ist (2,103), welche Regeln beachtet werden sollten und können Verbesserungsvorschläge für den vorgeschlagenen Code geben. Insofern erfüllt die Sozialisation eine wichtige Funktion und die Einführung in eine Rolle ergibt sich praktisch aus der Sozialisation durch das stetige Engagement. Bei der Größe der Entwickler-Community wird die Sozialisation mit den Werten der Community durch das Bewerbungsverfahren institutionalisiert, das zugleich auch ein Mittel der Selektion neuer Mitglieder darstellt. Aus Gründen der Nachvollziehbarkeit der angelegten Kriterien ist dieses Verfahren sehr genau auf den Webseiten beschrieben. Trotz der starken Verregelung hängt der Einstieg dabei aber sehr stark an der persönlichen Interaktion der Anwärter*innen mit den betreffenden *Debian Developers*. Die starke Selektion von Mitgliedern in den Kreis der Erwählten führt teils auch zu starken Bindungen innerhalb der Community, so nennt ein Entwickler als Motivation für das Engagement in Debian die Freund*innen und Menschen dort, die er zu schätzen gelernt hat: “... and a now, because

66 https://wiki.debian.org/DebianMentorsFaq#What_is_the_debian-mentors_mailing_list_for.3F, letzter Abruf 15.3.2017

I got many friends in the Debian project, so it's super nice when you know many people in the project and plenty of amazing folks over there" (2,36).

Formale Mitgliedschaft

Das Verfahren, um den Status *Debian Developer* zu erlangen, ist stark formalisiert. Für eine erfolgreiche Bewerbung muss man seine Fähigkeiten und Kenntnisse unter Beweis stellen und die eigenen Ziele und Visionen für das Projekt erklären.⁶⁷

"You have to pass, not an exam, but you have some questions about Free Software, you have to have technical skills that need to be tested and stuff like that." (11,17)

Unter den formalen Rollen in Debian wird (seit 2007) unterschieden in Paketbetreuer*innen mit eingeschränkten Zugriffsrechten, sogenannte *Debian Maintainer*, und Entwickler*innen mit vollen Zugriffsrechten, sowie Wahlrechten in Entscheidungsprozessen, die *Debian Developer*. Bewerber*innen für den Status des *Debian Developer* wird nahegelegt, zuerst *Debian Maintainer* zu werden und sich wenigstens ein halbes Jahr als Maintainer einzubringen.

Damit ist dem vollen Mitgliedsstatus eine Art Bewährungsstatus vorgelagert, der aber ebenfalls über ein formales Bewerbungsverfahren erlangt wird. In beiden Fällen brauchen Anwärter*innen einen *Debian Developer* als Mentor*in, der oder die deren jeweilige Beiträge und Leistungen für die Community durch vorherige Zusammenarbeit kennt und bewerten kann. Dies ergibt sich meist dadurch, dass aufgrund der fehlenden Zugriffsrechte externe Beitragende ihre Änderungsvorschläge zunächst an den *Debian Developer* schicken müssen, der für das Paket zuständig ist und die Änderungen einpflegt. Er oder sie wird auch *Sponsor* genannt und gibt gegebenenfalls in einer Art Peer-Review Verbesserungsvorschläge bezüglich der vorgeschlagenen Änderungen zurück. Dadurch erfahren Externe eine Sozialisation durch die Mitglieder der Community, die daher auch *Mentor* genannt werden.⁶⁸

Die Fürsprache eines Mentors oder einer Mentorin muss durchscheinen lassen, dass diese*r auch wirklich weiß, was der oder die neue Bewerber*in geleistet hat:

⁶⁷ Die Informationen über den Bewerbungsprozess speisen sich vornehmlich aus der Dokumentation auf der Website des Debian-Projekts, siehe <https://wiki.debian.org/DebianMaintainer> und <https://wiki.debian.org/DebianDeveloper>, letzter Aufruf 17.3.2017.

⁶⁸ <https://mentors.debian.net/>, letzter Aufruf, 20.3.2017

If the Debian Maintainer candidate has done “a great job”, please explain what “a great job” means – is there something special the candidate has done, or is it that whatever the candidate is working on is particularly important, or is the candidate remarkably consistent, or what?⁶⁹

Durch diese enge Betreuung, die einer Mitgliedschaft zur Voraussetzung gemacht wird, und die klare Beschränkung der Zugriffsrechte in der Zeit der Bewährung, wird ein hoher Qualitätsanspruch realisiert und sichergestellt, dass neue Mitglieder die notwendigen Fähigkeiten entwickeln und sich mit den Werten der Community identifizieren. Schließlich Bewerber*innen diese Fähigkeiten nachweisen und auch ihre Übereinstimmung mit den grundlegenden Dokumenten wie *Social Contract*, *Debian Free Software Guidelines* und *Machine Usage Policies* erklären, was im Laufe des Verfahrens geprüft wird.

Neben der fachlichen Kompetenz werden auch konkrete Zukunftsvorstellungen für das Projekt abgefragt und auf einen gepflegten Umgang mit anderen Entwickler*innen geachtet. Für eine erfolgreiche Bewerbung als Entwickler*in sollte ihr*e Mentor*in daher auch in einer Art Plädoyer über die Zukunftspläne und -visionen des Anwärters bzw. der Anwärtlerin für das Debian-Projekt und seine bzw. ihre Art der Interaktion mit anderen Usern und Mitgliedern der Community berichten.⁷⁰

Da der Kreis der Entwickler*innen mittlerweile zu groß dafür ist, dass sich alle Mitglieder persönlich kennen könnten, wird durch eine sozio-technisch implementierte Vertrauenskette („Web of Trust“) sichergestellt, dass wenigstens eine Reihe Entwickler*innen das neue Mitglied vis-à-vis gesehen haben. Dazu signieren diese Entwickler*innen mit einer kryptografisch eindeutigen Identität, ihrem GPG-Schlüssel, den Schlüssel des neuen Mitglieds. Dies ist ein fälschungssicheres Verfahren und über die Signierungen durch andere Debian-Entwickler weist jeder Schlüssel die sozialen Relationen des Besitzers bzw. der Besitzerin eines Schlüssels nach. Wer im Zuge dieses Verfahrens endlich den Status des *Debian Developer* erreicht hat, erhält dann die vollen Stimmrechte für die demokratischen Entscheidungsprozesse (ausführlicher s. 8.2.2).

Seit 2010 gibt es formal auch die Möglichkeit, *Debian Developer ohne Upload-Rechte* zu werden, wenn man ausreichendes Engagement durch nicht-technische Beiträge vorweisen kann. Dies deutet einen Wandel an,

69 <https://wiki.debian.org/DebianMaintainer>, letzter Aufruf 17.3.2017

70 <https://wiki.debian.org/DebianDeveloper>, letzter Aufruf 17.3.2017

spielt aber eine untergeordnete Rolle, insofern dies von keinem oder keiner Interviewten erwähnt wurde und diese mit 28 *Debian Developern* nur lediglich 2% der Entwickler*innen stellen (Stand 16.3.2018; ausführlicher s. 8.2.2).

Eine formale Selbstzuweisung wie in der Rolle des *Ubuntero* gibt es bei *Debian* nicht. In letzter Zeit hat aber die Bemühung, auch Beitragende, die keine formale Rolle innehaben zu würdigen, Ausdruck gefunden in der Schaffung einer Liste von *Debian Contributors*, die sich aus gesammelten Informationen über Aktivitäten in den spezifischen technischen Kollaborationsplattformen speist.⁷¹

8.1.3 Arch – „A Simple and Lightweight Distribution“

*RTFM helps the noob.*⁷²

Das adressierte Userbild von Arch Linux wird schon explizit auf der Projektseite ausgeführt:

Whereas many GNU/Linux distributions attempt to be more user-friendly, Arch Linux has always been, and shall always remain user-centric. The distribution is intended to fill the needs of those contributing to it rather than trying to appeal to as many users as possible. It is suited to anyone with a do-it-yourself attitude that's willing to spend some time reading the documentation and solving their own problems.⁷³

Hier findet sich also eine explizite Abgrenzung vom Paradigma der *Nutzerfreundlichkeit* oder *Usability* und der damit verbundenen Einstellung, möglichst viele User anziehen zu wollen. Demgegenüber richtet sich diese Distribution an diejenigen, die aktiv beitragen, eine „Do-it-yourself“-Attitüde haben und gewillt sind, etwas Zeit damit zu verbringen, die notwendige Dokumentation zu lesen und *ihre eigenen* Probleme *selbst* zu lösen.

Simple bedeutet in diesem Zusammenhang also nicht etwa „einfach und leicht zu bedienen“, sondern steht für eine Art der technischen Gestaltung, die auf eine einfache Wartbarkeit des Systems abzielt. Mit anderen Worten

71 <https://contributors.debian.org/>, letzter Aufruf 8.5.2017

72 <https://lists.archlinux.org/pipermail/arch-general/2012-May/026420.html>, letzter Aufruf, 24.3.2017

73 „Noob“ ist eine (abwertende) Kurzform für „Newbie“ und bedeutet Neuling; https://wiki.archlinux.org/index.php/Arch_Linux#User_centrality, letzter Aufruf 23.3.2017.

zielt das Wort *simple* ab auf ein System “without unnecessary additions or modifications”⁷⁴. Dieses Verständnis von *simple* ist Teil einer Philosophie, die auch „Keep it simple, stupid“, kurz KISS-Prinzip, genannt wird.⁷⁵ Einfachheit bezieht sich hierbei auf die Komplexität der Technik, die sich beispielsweise darin zeigt, dass die Konfiguration nicht über komplexe grafische Tools erfolgt, sondern ganz „simpel“ über einfache Text-Dateien.⁷⁶

Nutzerbild

Die Arch Community wendet sich deutlich an die „kompetenten und engagierten Nutzer“ (8,150). Mit dem formulierten Anspruch, „user-centric“ statt „user-friendly“ zu sein, fokussiert die Community ganz explizit auf User, die sich auch als Beitragende verstehen, und grenzt sich bewusst ab von der mit „Usability“ einhergehenden Zielsetzung, möglichst viele User zu gewinnen.

Vielmehr sind hier „Linux-Lerner“ (13,14) angesprochen, “those who wanna learn how stuff works” (10,51) – und User, die nicht bereit sind, sich selbst einzulesen und ihre eigenen Probleme zu lösen, „fallen in der Community durch“:

Wir haben halt ein bisschen den Vorteil, dass wir ne Distribution sind, die ziemlich klare Zielgruppen haben und nicht vom absoluten Anfänger bis zum Profi alles abdecken müssen. Wir können ganz klar sagen, OK, wenn du dich mit Arch beschäftigst, dann sind das so irgendwie Voraussetzungen, dann muss dir das und das gefallen, wenn es das nicht tut, ist auch egal, dann bist da aber hier eigentlich nicht richtig. (8,163f)

Voraussetzungen

Die Bereitschaft, lernen zu wollen, wie das System funktioniert (13,128), ist also eine Voraussetzung für die Teilnahme an der Community. Dass dies mitunter eine gewisse Hartnäckigkeit erfordert, spiegelt sich wider in der Aussage eines Entwicklers, der berichtet, dass er beim ersten Versuch der Installation scheiterte, weil es zu schwierig war, und sich erst beim zweiten Versuch Erfolg einstellte (1,54). Eine gewisse Erfahrung mit Linux ist ebenso hilfreich für die erfolgreiche Installation und Nutzung von Arch (8,150). Die

⁷⁴ https://wiki.archlinux.org/index.php/Arch_Linux#Simplicity, letzter Aufruf 23.3.2017

⁷⁵ Vgl. https://wiki.archlinux.org/index.php/Arch_terminology#KISS bzw. <https://de.wikipedia.org/wiki/KISS-Prinzip>, siehe auch https://de.wikipedia.org/wiki/Arch_Linux, letzter Aufruf 23.3.2017.

⁷⁶ Die technische Einfachheit des Installationsprozesses wird in Abschnitt 9.3 deutlich.

Nutzung von Arch kann also verbunden sein mit einem deutlichen Zeitaufwand, nicht zuletzt, weil die starke Priorität der Aktualität es mit sich bringt, dass durch häufige Updates auch die Konfiguration häufig angepasst werden muss (13,128): “I use it a lot as a learning tool, I mean, there is a lot of overhead, you have to have free time to run it, because you constantly update things and change configs, and it’s just lot of your time” (4,128).

Grafische Tools zur Konfiguration gibt es zwar mitunter, aber für den Support durch die Community wird erwartet, dass jede*r auch die Kommandozeile bedienen kann und auch die dort abrufbaren Anleitungen, sogenannte *Man-Pages*, zu Rate zieht, bevor er oder sie Fragen über die Support-Kanäle stellt (8,128ff). Dies ist mitunter der Gegebenheit geschuldet, dass bei einer direkten Eingabe eines Befehls eine unmittelbare Rückmeldung mit Fehlermeldungen erfolgt (oder dies leicht mit entsprechenden Befehlsparametern erreicht werden kann). Somit findet eine direktere Interaktion (KISS-Prinzip) mit dem Computer statt, was hilfreich für die Fehlersuche und damit die Betreuung durch räumlich distanzierte Mitglieder der Community ist. Die Text-Ausgabe der Kommandozeile kann folglich leicht über die (textbasierten) Support-Kanäle übermittelt werden und ist für unterstützende Arch-User leichter nachvollziehbar als die Beschreibung der Reaktion fensterbasierter, grafischer Anwendungen, wo die Fehlermeldungen häufig weniger aussagekräftig sind.

Da für eine erfolgreiche Nutzung des Systems eine Bereitschaft bestehen sollte, die Anleitungen und *Man-Pages* der jeweiligen Systemkomponenten zu lesen, kann die minimale Voraussetzung hier als *Primary Source Knowledge* gefasst werden (vgl. Abschnitt 5.1).

Einstieg

Entsprechend der hohen Anforderungen an die Fähigkeiten der User, gewisse Zusammenhänge im System zu erkennen und sich mittels Kommandozeile und *Man-Pages* zunächst selbst mit aufkommenden Problemen zu beschäftigen, fallen User, die Support als ihr „gutes Recht“ verstehen, durch:

Also eigentlich wollen wir ja die kompetenten und engagierten Nutzer haben, ... und wir möchten auch, dass die Nutzer im Prinzip Teil der Entwicklung sind. Dass sie auch sich dessen bewusst sind, dass wir nicht alles vorfertigen, und dass wenn irgendwas nicht geht, dass wir für keinen sagen – wir machen das dann alles – sondern dass jeder selbst mitarbeiten soll. Ja, das sind so unsere Win-Schubser und ja, es gibt sicherlich auch andere, ((lachen)) ja, die fallen wahrscheinlich auch so in der Community durch. (8,150–152)

Der Begriff „Win-Schubser“ ist hierbei eine Mischung aus „Maus-Schubser“ und „Windows User“, die impliziert, dass die betreffenden User sich nur auf der grafischen *Oberfläche* bewegen können und somit die *Tiefen* des Systems gar nicht begreifen oder per Kommandozeile erschließen können. Andererseits wird dieser Kategorie angekreidet, dass sie als Anwender*innen ein gewisses Selbstverständnis, dass Support ihr Anrecht ist und ein Entgegenkommen seitens der technisch-versierten Unterstützer*innen erwartet, mitbringen.

Dabei kollidieren hier zwei besonders gegensätzliche Technik-Verständnisse unterschiedlicher Nutzertypen miteinander; Experten-User und Laien-User, die in ihrem Zugriff auf Technik unterschiedlich sozialisiert sind. Die einen gehen selbstverständlich von einem Service aus, der ihnen als Laien gegenüber geschuldet wird, und die anderen gehen ebenso selbstverständlich davon aus, dass man, bevor man sich die Blöße gibt, eine triviale Frage zu stellen, auf alle verfügbaren Wissensquellen zurückgreift und versucht, das Problem selbst in den Griff zu bekommen. Vor dem Hintergrund der Fokussierung auf engagierte und kompetente, potenziell beitragende User ist es aus Sicht der Community daher auch nicht weiter dramatisch, wenn User die diese Voraussetzungen nicht erfüllen, „hinten runterfallen“.

Das heißt aber nicht, dass Fragen stellen an sich illegitim ist, es wird auch gerne Auskunft gegeben, wo sich die Antwort finden lässt, nur wer diese Hinweise übergeht und nicht erkennen lässt, dass er oder sie sich Mühe gibt, das Problem aktiv zu lösen, bekommt eine Abfuhr. Selbst der kurz angebundene Verweis „RTFM – Read the Fucking Manual“ ist in diesem Verständnis ein großzügiger Hinweis auf Defizite – großzügig, weil das Gegenüber sich überhaupt die Zeit nimmt zu antworten, und diesen Hinweis idealiter besonders dann verwendet, wenn die notwendigen Informationen (aus Sicht des oder der Antwortenden) offensichtlich sind oder einfach zu finden gewesen wären.

Wenn man freundlich ist, gibt man ihm kurz einen Hinweis, welches Package er installieren muss, und von da aus kann man ihm ja sagen, ok, dann schau ins Wiki. Dann ist man das Problem los, dem ist geholfen, und er weiß in etwa, wo er nachschauen soll, und wenn er dann ein funktionierendes Netzwerk zwar hat und einen Firefox, oder was für einen Browser er auch immer nimmt, aber nicht ins Wiki geht, sondern direkt weiter fragt, dann ist der bei mir abgewählt, dann kriegt der vielleicht noch zwei, dreimal von mir gesagt, geh ins Wiki. (8,172–174)

Wenn ein unerfahrener User Pech hat, kann auf eine einfache Frage auch mal mit einer „nutzlosen“ Antwort reagiert werden, wenn im Support-Kanal des Internet Relay Chat (IRC) ein paar Spaßvögel sind (8,210ff) – je nachdem, „welche erfahrenen Nutzer gerade wo mitlesen und wie deren *Laune* ist“ (8,198). Dies ist eigentlich unerwünscht, kann aber im IRC zwischendurch mal geschehen, wenn sich entsprechende Leute dort befinden. Diese Aussage wird ergänzt durch eine Klage über eine sinkende Qualität der Beiträge in der Dokumentation und im Wiki, die in Zusammenhang gebracht wird mit der zunehmenden Popularität von Arch Linux. Die elitäre Exklusion von Laien hat also offensichtlich eine Funktion für die Selektion der Beitragenden innerhalb der Community. Die erwähnte Unart, nutzlose Antworten zu geben, wird innerhalb der Community auch nicht allzu problematisch gesehen, weil ohnehin jeder User zum Mitdenken verpflichtet ist und gegebenenfalls selbst den Unfug bemerken sollte:

Generell wenn ich als Nutzer eine Antwort kriege, über welchen Kanal auch immer, sollte ich das jetzt nicht blind umsetzen, mir das durchlesen, und ich sollte auch versuchen zu verstehen, was ich da tun soll. Weiß ich nicht, ob das jeder macht, aber das wär' *wünschenswert*, wenn man das tun würde. (8,249)

Hinzu kommt eine (in ganz sachlichem Sinne) eigennützige Verwendung des Systems, die eben nicht auf Sendungsbewusstsein und „Advocacy“ gepolt ist und sich so ganz anders als im Fall von Ubuntu oben darstellt:

Ein Großteil meines Engagements läuft darauf hinaus, dass ich das System selber nutze, und das ist dann eher ein egoistischer Ansatz, würde ich mal behaupten ... es ist nicht so, dass ich aktiv immer die Foren und Mailinglisten abgrase und schaue, wo kann ich jemandem helfen? Es ist, wenn ich abends dann mal die Mailinglisten durchlese, dann ist den Meisten schon geholfen. (8,184–186)

Diese Aussage stammt von einem Entwickler, der sich aktiv in der Entwicklung engagiert, die User-Betreuung ist aber in dieser Perspektive ein Nebenprodukt. Dies gilt zwar in der ein oder anderen Form auch in in anderen Communities, unterstreicht aber hier die Erwartung gegenüber Usern, sich nötigenfalls selbst zu helfen, und illustriert eine pragmatische, eigennützige Einstellung zum FLOSS-Prinzip – im Gegensatz zu einer entgegenkommenden, advokatischen Einstellung, die in Ubuntu an vielen Stellen prominent ist.

In diesem Zusammenhang ordnet sich das Akronym RTFM ein als ein sachlicher Hinweis darauf, dass Fragende hier noch etwas mehr recherchieren und nachlesen sollten. Dies wird in folgendem Auszug auf der Mailingliste diskutiert: In einer Mailinglisten-Diskussion auf der Arch-User-Liste

fragt ein offensichtlicher Laie, wie er mit einem Programm YouTube-Videos herunterladen kann. Diese Frage ist in verschiedener Hinsicht deplatziert: Einerseits verwendet der User offensichtlich eine von Arch völlig verschiedene, explizite Einsteigerdistribution (Linux Mint ist ein auf Ubuntu basierendes System), andererseits gibt sich der User scheinbar wenig Mühe, einen vollständigen Satz zu formulieren und spricht mit dem Download von YouTube-Videos obendrein ein rechtlich heikles Thema an: “How to download youtube video using wget. I am a new user. i am using linux mint 12.”⁷⁷

Auf den recht höflichen Hinweis darauf, dass die Liste der Arch-Distribution das falsche Forum für eine Frage zu Linux Mint ist, stellt der User zwei Fragen. Die erste Frage zielt darauf ab, ob die Befehle in diesem Fall bei Arch anders als bei Linux Mint seien – im Grunde eine berechnete Frage, denn wget ist ein altbewährtes universelles Tool, das im Grunde auf allen Unix-Systemen mehr oder weniger gleich funktionieren sollte. Daraus folgt, dass mit wenig Aufwand viele Dokumentationen und Anwendungsbeispiele zu finden sind. Darüber hinaus verkennt die Frage den impliziten Hinweis, dass vor allem das Forum das falsche ist und daher weitere Fragen unerwünscht sind, was freilich für den unbedarften Fragenden nicht ohne weiteres erkennbar ist. Arch Linux richtet sich eben an User, die sich erstmal selbst schlau machen und nicht geradewegs jede aufkommende Frage erstmal an andere stellen. Distributionen mit Usability-Fokus, wie Linux Mint, erscheinen hierbei als Negation der selbsttätigen Aneignung von Wissen. Die Frage nach der Mailingliste von Linux Mint wirkt hierbei schließlich als Provokation.⁷⁸ Die Antwort fällt vor diesen Hintergrund in ihrem lakonischen Zynismus im Grunde relativ freundlich aus: “Yes, drastically different”⁷⁹ (s. Abb. 8.1).

77 <https://lists.archlinux.org/pipermail/arch-general/2012-May/026397.html>, letzter Aufruf 24.3.2017

78 Es ist auch gut möglich, dass die Frage als böswilliger Spaß gemeint sein könnte, um Arch-User zu ärgern. Es ist für manche sogenannte „Trolle“ ein regelrechter Sport, mit deplatzierten Fragen und Bemerkungen, Streit zu stiften. In jedem Fall entfaltet der Beitrag hier eine interessante Wirkung.

79 <https://lists.archlinux.org/pipermail/arch-general/2012-May/026403.html>, letzter Aufruf 9.5.2017

```

On Wed, May 02, 2012 at 12:32:43PM +0530, [Mint User] wrote:
>>> how to download youtube video using wget . I am a new user. i am
>>> using linux mint 12.

On Wed, May 2, 2012 at 12:50 PM, [Arch User] wrote:
>> You go to Arch Linux mailing list to ask a question about youtube and
>> wget, and you're using linuxmint? :|

On 05/02/2012 12:25 AM, [Mint User] wrote:
> are the commands are different for archlinux and linuxmint?
> i dont know, if it is so sorry for disturbing you.can u say what is
> mailling list for linuxmint?

[...]

Yes drastically different

```

Abb. 8.1 Arch-User-Diskussion auf der User-Liste

Die bewusst sachlich falsche Aussage des Antwortenden tritt eine Diskussion über die Legitimität von falschen Hinweisen los. Dies mündet in der Feststellung eines Users, dass es legitim sei, jemanden gegebenenfalls auch ironisch auf die Deplatzierung seiner Frage hinzuweisen und ihm mit RTFM zu antworten, es aber dennoch inadäquat erscheine, eine konkrete *falsche* Antwort zu geben.⁸⁰

- > Honestly to me if you are using a different OS, make no claim to
- > using Arch and you decide to post to here to find an answer,
- > a poking is well deserved.

A poking, or an RTFM, sure. All I'm saying is that what you posted doesn't help the noob. RTFM helps the noob. Pointing out that this place has nothing to do with Mint, even mockingly (even at Drepper levels of intensity), helps the noob.

“Making a joke” that is only a joke to people who know enough not to do the thing the noob did does not help the noob.

It's not your responsibility to help, it's also not your responsibility to hinder. We were all dumb noobs once, and I for one don't feel good if I plant misinformation in someones head, intentionally or not.

Es wird also klargestellt, dass es nicht in die Verantwortlichkeit der User der Liste fällt, zu helfen – aber eben auch nicht, andere zu behindern.

⁸⁰ <https://lists.archlinux.org/pipermail/arch-general/2012-May/026420.html>, letzter Aufruf, 24.3.2017

Der Einstieg in die Community bezieht sich hier aufgrund des Nutzerbildes zugleich auf die Nutzung als auch auf die aktive Kontribution. Anders als bei Ubuntu steht hier aber die technische Kontribution im Vordergrund und Dokumentation erscheint als eine selbstverständliche Pflichtübung – ebenso, wie die Dokumentation zu Rate zu ziehen, bevor man eine Frage stellt. Sie ist im Gegensatz zum Fall Ubuntu also weniger eine bewusste Vereinfachung des notwendigen Wissens, um möglichst viele anzusprechen, sondern es ist klar, dass Einsteiger*innen eine steile Lernkurve zu absolvieren haben.

Schließlich ist man als User auch angesprochen, Software-Pakete ins allgemein zugängliche *Arch User Repository*, kurz AUR, zu stellen, und kann damit auch Entwicklertätigkeiten übernehmen. Im AUR haben User folglich – neben den anderen Beitragsformen des Bug Reporting, Dokumentation etc. – auch die Möglichkeit, sich als engagierte User und Paket-Verwalter*in zu erweisen. Das eigenverantwortliche Anbieten fehlender Software im AUR bietet also eine Möglichkeit, in die Verwaltung des Systems einzusteigen und sich gegenüber den Entwickler*innen zu beweisen, als gewissenhafter User mit Engagement und Ambitionen (mehr dazu im folgenden Abschnitt und in Abschnitt 8.3.3).

Formale Mitgliedschaft

Die Rolle des *Trusted User* repräsentiert eine formale Rolle. Bemerkenswerterweise ist das Bereitstellen von Software-Paketen nicht den formalen Mitgliedern vorbehalten, sondern es gibt ein allgemeines *Arch User Repository* (AUR), auf dem jeder User Software-Pakete bereitstellen darf. Die *Trusted User* haben aber die Kontrolle über das AUR und können beliebige Pakete in das offizielle *Community*-Repository überführen.

Für eine Mitgliedschaft als *Trusted User* kann man sich bewerben, indem man sich an die *Trusted-User-Mailingliste* wendet. Ähnlich wie bei Debian braucht man als Bewerber*in eine*n Unterstützer*in, jedoch ist das Verfahren weniger formalisiert. Nach der Bewerbung stimmen die anderen *Trusted User* über den Antrag ab und in aller Regel – wenn keine erheblichen Gründe entgegenstehen und Gegenstimmen erhoben werden – wird dies gestattet. Darüber hinaus müssen dann drei *Arch Developer* den öffentlichen GPG-Schlüssel des neuen *Trusted User* signieren, damit dieser auch Upload-Rechte bekommt (8,281). Das Vertrauen der *Arch Developer* wird also hier über ein technisches Authentifizierungsverfahren ausgesprochen.⁸¹

81 Hier liegt ein asymmetrisches Verschlüsselungsverfahren mit zugrunde.

Für den schreibenden Zugriff auf das *Arch User Repository* muss man sich lediglich einen Account im AUR anlegen und einen öffentlichen Schlüssel hinterlegen, mit dem man sich authentifizieren kann.⁸² Dies ist aber jenseits der Verantwortlichkeit für die eigens angebotenen AUR-Pakete und mit keinen besonderen formalen Rechten oder Zuständigkeiten innerhalb der Community verbunden. Dafür bekommt man aber keine besondere Rollenbezeichnung wie bei Ubuntu's „Ubuntero“-Rolle, sondern bleibt einfach ein „Arch User“.

Die niederschwellige Rolle, die durch eigentätige Registrierung möglich ist, bringt schon Entwicklungsbefugnisse mit sich, ist namentlich aber eine User-Rolle. Darüber hinaus bedingt die Rolle des Trusted User mit weiterreichenden Rechten zwar ein formales Aufnahmeverfahren, wird namentlich aber ebenfalls als User bezeichnet. Dies illustriert, dass in Arch User als Mitentwickler*innen betrachtet werden.

8.1.4 Vergleich der Variable „Mitgliedschaft“

Das kollektive Selbstbild der Community wird beschrieben durch die implizite und explizite soziale Konstruktion der Mitgliedschaft. Das Konstrukt der Mitgliedschaft rahmt nicht nur den Zugang zur Nutzung des Systems, sondern strukturiert auch die aktive Teilnahme an der Kontribution. Diese ist zwar potenziell immer möglich, wird aber unterschiedlich stark als Teil der User-Rolle betrachtet. Die Validität von Beiträgen wird dabei ebenfalls unterschiedlich bewertet; insofern unterscheiden sich nicht nur Voraussetzungen, die an die Nutzung gestellt werden, sondern auch die an das Beitragen gestellten Anforderungen. Vor dem Hintergrund der technik-typischen Wissensdifferenz zwischen Expert*innen und Laien fällt hierbei auf, dass die unterschiedlichen Erwartungen von Expertise sich in der Ausrichtung der Community auf unterschiedliche Nutzerbilder widerspiegeln.

Die Ergebnisse der Auswertung werden im Folgenden pointiert zusammengefasst und in Tabelle 8.1 gegenübergestellt. In der tabellarischen Darstellung habe ich die Eingangszitate zum Akronym RTFM als RTFM-Policy

⁸² https://wiki.archlinux.org/index.php/Arch_User_Repository#Authentication, letzter Aufruf 9.5.2017. In diesem Fall handelt es sich um einen SSH-Key, technische Details spare ich an dieser Stelle aus und verweise auf die *Man-Page* (`$ man ssh`), ebenfalls sehr informativ ist hier Wikipedia, https://de.wikipedia.org/wiki/Secure_Shell, letzter Aufruf 9.5.2017.

aufgenommen, da diese die Gesamttendenz der jeweiligen Ausprägungen der Variable veranschaulichen.

Tab. 8.1: Pointierter Vergleich der Variable Mitgliedschaft

Dimension	Ubuntu	Debian	Arch
<i>Read The Fucking Manual</i> „Policy“	<i>RTFM ist nicht regelkonform</i>	<i>eine angemessene Antwort und Appell, die Dokumentation zu verbessern</i>	<i>ein großzügiger Hinweis auf Defizite</i>
Nutzerbild	„telly user“	„serious, technically-capable Linux users“	„those contributing to it“
Voraussetzungen	<i>Beer-Mat Knowledge</i> – wenig Vorwissen für Installation, Verwendung und eine aktive Beteiligung notwendig.	<i>Popular Understanding</i> – grundlegendes Wissen über das Systems, für eine aktive Rolle: Regeln und Policies beachten	<i>Primary Source Knowledge</i> – „Read the Fucking Manual“
Einstieg	lokale Anwender-treffen	Mentoren	<i>Arch User Repository</i>
formale Mitgliedschaft	Signierung des <i>Code of Conduct</i> (Ubuntero), explizit nicht-technisches Ubuntu Membership	zweistufige Bewerbung: erst <i>Debian Maintainer</i> , dann <i>Debian Developer</i> mit Stimmrecht	AUR-Account selbst anlegen, <i>Trusted User</i> -Bewerbung

RTFM wird in Ubuntu als nicht konform mit den gemeinsamen Regeln der Freundlichkeit erachtet (*Code of Conduct*), während in Debian RTFM einen sachlichen Appell darstellt, bei Unklarheiten tiefer in die Materie einzusteigen und gegebenenfalls die bislang als unzureichend befundene Dokumentation zu ergänzen und zu verbessern. In Arch hingegen ist das Akronym ein Hinweis auf Defizite des Users, der froh sein sollte, überhaupt eine Antwort auf eine triviale Frage zu bekommen anstatt ignoriert zu werden.

Nutzerbilder und (minimale) Voraussetzungen

Ubuntu Ubuntu ist mit dem Fokus auf „Human Beings“ laut Interviewausgabe auch für „Fernseh-Nutzer“ nutzbar (13,128). Fernseher stehen dabei für eine klassische Technik, die – ausgestattet mit einer überschaubaren Anzahl von Bedienknöpfen – keinerlei Kenntnis über die Funktionsweise der Fernsichttechnik voraussetzt. Dementsprechend ist Usability hier ein leitender Wert, der eine „intuitive“ Nutzung mit möglichst wenig Hintergrundwissen ermöglichen soll. Folglich sind die Voraussetzungen für User und aktiv Beitragende recht niedrig angesetzt. Sowohl für die Nutzung als auch beispielsweise für die Beitragsform *Advocacy* reichen ein anwendungsbezogenes Wissen und ein eher schematisches Verständnis der Funktionsweise der Software – *Beer-Mat Knowledge*, ist hier also ausreichend.⁸³

Debian Debian spricht mit seiner Ausrichtung auf Stabilität und Universalität eher Power-User und Administrator*innen mit technischem Verständnis an, die zwar unkomplizierte Tools wie Installationsroutinen und eine Paketverwaltung schätzen, aber auch ein Bedürfnis haben, das System für ihre spezifischen Bedürfnisse anzupassen und zu konfigurieren. Dementsprechend sollten User etwas mehr als ein rein schematisches Wissen mitbringen und schon mal etwas über die grundsätzliche Funktionsweise der Rechner gelesen haben – etwa dass die Hardware, um zu funktionieren, Treiber braucht, denn gegebenenfalls muss man diese separat in die Installation einbinden (vgl. S. 132). Für eine aktive Rolle in der Community sollte man auch schon Mailinglisten bedienen können und idealerweise auch ein Terminal. Ein minimales Verständnis dafür lässt sich über die Lektüre von einschlägigen Computer-Zeitschriften anlesen. Das benötigte Vorwissen fällt daher in die Kategorie *Popular Understanding*.

Arch Arch wird selbst von Usern als „Lern“-Linux bezeichnet und adressiert explizit User, die zur Distribution beitragen („User centrality“). Lernbereitschaft und die Auseinandersetzung mit Bedienungsanleitungen (RTFM) sind also zentrale Voraussetzungen. *Primary Source Knowledge* begründet somit die Voraussetzung für Nutzung und für den Einstieg in eine aktive User-Rolle. Der Grundsatz „solve your own problems“ bezieht sich dabei von der Lösung von Konfigurationsproblemen auf Anwendungsebene bis hin zum Beitrag von Code im Upstream.

83 Für die verschiedenen Abstufungen von Wissen und Expertise vgl. Abschnitt 5.1.

Einstieg und formale Mitgliedschaft

Da sich die Beitragenden einer Community aus ihren Usern rekrutieren, bedeutet Einstieg in die Mitgliedschaft sowohl Einstieg in die Nutzung als auch Einstieg in eine aktive Rolle als Beitragende, die potenziell immer vorgesehen ist. Die Einstiegspfade in eine aktive User-Rolle in der Community setzen somit an bei der Nutzung und münden nicht zwangsläufig in einer aktiven Rolle. Hier unterscheiden sich die Einstiegspfade in die Community sowie die Anforderungen an die formale Mitgliedschaft erheblich.

Es gibt in allen Fällen Mitgliedsverfahren, in deren Verlauf die Bewerber*innen Engagement beziehungsweise Erfahrung und Kompetenz vorweisen müssen und eine*n Fürsprecher*in aus dem Kreis der schon bestehenden Mitglieder benötigen. Die Fälle variieren hier in der Bedeutung der Rolle einer solchen Mitgliedschaft und der Kriterien, die angelegt werden.

Ubuntu Ubuntu wird auch von erfahrenen Anwender*innen als Einstiegsoption für Freunde und Bekannte betrachtet, die hier mit wenig Vorwissen einsteigen können. Die Fokussierung auf Usability wirkt somit als institutionalisierter Einstieg auch in eine aktive Rolle. Darüber hinaus sind lokale Anwendertreffen, sogenannte *Meet-ups* und User-Stammtische, von Bedeutung. Hier können Erfahrungen ausgetauscht werden und Hilfe zur Selbsthilfe kann angeboten werden.⁸⁴

Diese Öffnung gegenüber einfachen Usern spiegelt sich wider in der formalen Mitgliedschaft. Für eine erfolgreiche Bewerbung zum *Ubuntu Member* ist zwar der Nachweis von Engagement notwendig, dazu zählen aber explizit auch Beitragsformen, die nicht technischer Art sind. Die Moderation in Diskussionen oder Hilfe bei Veranstaltungen wie *Meet-ups* sind also explizit valide Beitragsformen, die für eine Mitgliedschaft qualifizieren. Noch niederschwelliger kann jeder User sein Commitment zur Community durch eine digitale Signierung des *Code of Conduct* ganz selbstständig bekunden.

Laien, also User, die nur ein schematisches Verständnis des Systems haben (*Beer-Mat Knowledge*), werden hier somit als User angesprochen und auch explizit als potenziell Beitragende betrachtet und haben die Möglichkeit, formales Mitglied zu werden.

⁸⁴ Ähnliches gab es schon vor „Ubuntu“ in Form von Linux User Groups (LUGs), fand aber in den Interviews keine Erwähnung. LUGs gibt es aber weiterhin und einige Ubuntu-Stammtische haben sich in Distributions-unspezifische Linux User Groups gewandelt oder haben sich mit ihnen zusammengeschlossen.

Debian Debian bietet ausführliche Installationsanleitungen⁸⁵ und schlägt eine Reihe von Konfigurationen im Installationsprogramm vor, um auch für User mit weniger Kenntnissen einen Einstieg zu bieten. Ziel ist aber dabei, ein gewisses Verständnis des Systems zu erlangen. In eine aktive Rolle als Beitragende gelingt der Einstieg am besten über den direkten Kontakt mit *Debian Developern*, was für eine formale Rolle ohnehin notwendig ist. Hierbei ist das Verständnis von Beiträgen noch stark geprägt von der Mitarbeit an der Software, ein Wandel hin zur offenen Wertschätzung auch weniger technischer Beiträge ist aber zu beobachten.

Für die formale Mitgliedschaft ist ein aufwendiges Bewerbungsverfahren notwendig, in dem technische Expertise des Systems und organisatorische Kenntnisse der Community abgefragt werden. Zunächst erwirbt man begrenzte Rechte als *Debian Maintainer*, um dann nach einer Zeit der Bewährung ein vollwertiger *Debian Developer* zu werden. Die Regeln und Policies der Community haben hier eine stark selektive Wirkung. Trotz der großen Anzahl von Mitgliedern wird durch die Regeln und das Bewerbungsverfahren eine hohe Kohärenz und Qualität angestrebt, dabei sind die Kriterien explizit und transparent.

Der Einstieg in die Community impliziert die Aneignung von Wissen. Für eine formale Rolle muss (üblicherweise) technisches Verständnis vorhanden sein; prinzipiell ist dafür also *Contributory Expertise* notwendig, da für eine erfolgreiche Bewerbung Code-Beiträge vorgewiesen werden müssen. Formal können seit 2010 zwar auch nicht-technische User *Debian Developer* werden, mit unter 3% ist deren Anteil aber noch marginal – hier zeigt sich aber ein gewisser Bewusstseinswandel bezüglich der Bedeutung der Beiträge jenseits der Software-Entwicklung (vgl. S. 165).

Arch Der Einstieg in die Nutzung erfolgt bei Arch durch die Kenntnis oder Aneignung der Konfigurationswerkzeuge auf der Kommandozeile (RTFM), die schon für eine erfolgreiche Installation notwendig sind. Über die Lektüre der Dokumentation im Laufe der Installation und der Lösung der eigenen Probleme erhält man auch potenziell Schritt für Schritt ein zunehmendes Verständnis des Systems – aus *Primary Source Knowledge* wird durch die Auseinandersetzung mit dem System und der Interaktion mit Entwicklern eine *Interactional Expertise*. Eine wichtige Rolle spielt das *Arch User Repository* (AUR), in dem jeder User durch die Bereitstellung von

85 Vgl. <https://www.debian.org/releases/stable/installmanual>, letzter Aufruf 24.3.2018.

Software Entwickleraufgaben übernehmen kann und somit *Contributory Expertise* erlangen kann.

Diese ist wiederum notwendig für eine formale Rolle als *Trusted User*. Diese rekrutieren sich im Wesentlichen aus den im AUR aktiven Usern, die Pakete bereitstellen. Als *Trusted User* bekommen sie von den Entwickler*innen ihr Vertrauen ausgesprochen (durch die Signierung ihrer GPG-Schlüssel) und führen Aufsicht über das AUR. Schließlich fügen sie die beliebtesten Pakete dem *Community-Repository* hinzu und machen sie somit zum Teil der Distribution.

Die Schwelle für eine Mitgliedschaft ist also in Ubuntu äußerst niedrig und integriert explizit Nicht-Entwickler*innen, aber der aus dieser Rolle resultierende Einfluss bleibt gering. Im Gegensatz dazu ist eine formale Rolle in Debian in Form eines *Debian Developers* eine Rolle mit wesentlichen Mitentscheidungsmöglichkeiten. Die *Trusted User* in Arch haben immerhin einen Einfluss auf die erweiterte Software-Basis, die Rolle ist aber durch und durch auf eine Unterstützung des Projekts durch Entwicklertätigkeit ausgerichtet und die Mitbestimmung ist begrenzt auf die Selbstorganisation und die Verwaltung ihrer Repositories.

8.2 Strukturen der Community

Generell gibt es in der Welt der Open Source Software einen Grundsatz, den ein Interviewter „Do-ocracy“ nannte: Wer etwas macht, der hat auch die Macht (11,139): „Von daher, ich glaub, ganz viele Sachen werden einfach nicht entschieden, die werden gemacht ... Also es ist ja so, dass auch ein Wahnsinniger Macht hat, um diese Software anders zu entwickeln“ (15,142). Dies deckt sich auch mit den Befunden von Tauberts Studie zum KDE-Projekt (vgl. Taubert 2006), der als ein Element in der Entscheidungsfindung die Schaffung von Fakten durch aktive Programmierung identifiziert.

Außerdem ist ein interessantes Mächtigegleichgewicht darin zu beobachten, dass jederzeit jede*r die Freiheit hat, bei Unstimmigkeiten den Code zu duplizieren und damit ein neues Projekt zu gründen, einen sogenannten „Fork“ zu machen. Das entlastet einerseits die Community davor, allzu große Kompromisse einzugehen, da bei starken Differenzen immer auch die Möglichkeit im Raum steht, das Projekt zu verlassen. Andererseits stellt die Fork-Mög-

lichkeit auch eine latente Gefahr für die Projektleitung dar, da, wenn eine kritische Masse an unzufriedenen Entwicklern erreicht ist, diese sich leicht abspalten können und damit deren Arbeitskraft verloren geht. Die Freiheit zu „forken“, wenn etwas nicht passt, spielt aber für Laien zunächst eine geringe Rolle, da sie gar nicht die Möglichkeit haben, die Software eigenhändig zu verändern.

Im Allgemeinen ergeben sich aus dem Forken von Projekten große Nachteile, da in jedem Teilprojekt nur eine verminderte Anzahl von Entwickler*innen mitarbeitet. Zwar können auch parallele „Gabelzweige“ voneinander profitieren, aber dies geht mit einem erheblichen Mehraufwand einher, da Änderungen in den verschiedenen Zweigen gegenseitig eingearbeitet werden müssen. Daher ist der Fork grundsätzlich eine suboptimale Lösung. Die Möglichkeit des Forks verkörpert das geflügelte Wort des „rough consensus“. Ein grober Konsens ist nötig, um weiter ein gemeinsames Ziel zu verfolgen – aber was über einen groben Konsens hinausgeht, sind hinderliche Spielereien, die vernachlässigt werden.

Um den Konsens zu erreichen und zu erhalten und eine Spaltung zu vermeiden, gibt es aber in allen größeren Communities Entscheidungsstrukturen und Verfahren. Durch interne Diskussionen und Abstimmungsprozesse werden die Interessen der freiwillig mitarbeitenden Entwickler*innen in Betracht gezogen und damit die Entscheidungen legitimiert. Im Hinblick auf diese Strukturen unterscheiden sich die Communities erheblich, wie ich im Folgenden herausarbeite.

Für die Erhebung wurde abgefragt, welche Rollen typischerweise in Entscheidungen eingebunden sind und welche vergangenen Entscheidungen oder Diskussionen den Befragten im Gedächtnis waren, um am praktischen Beispiel mehr über die Funktionsweise der Organisation und die Strukturen der Community zu erfahren.

Bei der Auswertung wurden folgende Dimensionen besonders berücksichtigt: Wer sind die Akteure, die an Entscheidungen beteiligt werden, wie stellt sich deren Entscheidungsraum dar, wie wird ihre Autorität legitimiert, welche Kriterien der Entscheidungsfindung werden zugrunde gelegt und welche Entscheidungsmodi und institutionalisierte Verfahren spielen eine Rolle. Die Strukturierung des Kapitels erfolgt entlang der Akteursgruppen, die zu Beginn jedes Abschnitts auch in einem schematischen Diagramm dargestellt werden. Die Rollen und Entscheidungsprozesse dienen der Legitimation von Entscheidungen und stiften durch die Institutionalisierung der Governance schließlich auch nutzerseitiges Vertrauen in die Software. Zudem ergeben

sich aus den Strukturen auch Einflussmöglichkeiten verschiedener Akteure auf die Gestaltung der Software, die je Community unterschiedlich ausgestaltet sind.

Da sich alle Open Source Communities als Meritokratien verstehen, habe ich für die Überschriften andere Herrschaftsformen gewählt, die die jeweilige Meritokratie spezifizieren.

8.2.1 Ubuntu – zwischen Community und Unternehmen

Ein substanzieller Unterschied zu den anderen beiden Fällen ist, dass Ubuntu vom Unternehmen Canonical ins Leben gerufen wurde und maßgeblich gesponsert wird. Mark Shuttleworth, der Gründer des Unternehmens, ist daher bis heute eine zentrale Figur des Projekts Ubuntu und hat eine Rolle als „Self Appointed Benevolent Dictator For Life“ (SABDFL). Dennoch gibt es im Unterschied zu anderen unternehmensgetriebenen GNU/Linux-Distributionen keine Unterscheidung zwischen kommerzieller Distribution und „Community-Edition“, sondern die gesamte Ubuntu-GNU/Linux-Distribution ist frei erhältlich, Canonical finanziert sich vor allem durch verwandte Dienstleistungen.

Von Anfang an tritt Ubuntu mit einer Agenda in Erscheinung, mit viel Kapital die Linux-Landschaft für den „normalen User“ zugänglich zu machen und die Hegemonie von Microsoft anzufechten („Bug #1: Microsoft has a majority market share“)⁸⁶. Die Finanzierung durch Mark Shuttleworth ermöglichte eine erfolgreiche Produktentwicklung durch Marketing, Design und nicht zuletzt Community-Management.

In Abbildung 8.2 sind die wesentlichen Organe der Community abgebildet.⁸⁷ Der SABDFL nimmt in der Community eine zentrale Rolle ein, da er die zwei wesentlichen Entscheidungsgremien *Technical Board* und *Community Council* ernennt. Im *Technical Board* hat er außerdem einen ständigen Sitz und eine gewichtete Stimme. Diese Gremien werden aber von den

⁸⁶ Siehe <https://bugs.launchpad.net/ubuntu/+bug/1>, letzter Aufruf 17.5.2017.

⁸⁷ Es gibt noch weitere Gremien, die spezialisierte Bereiche haben, die aber im Prinzip den Hauptgremien *Technical Board* und *Community Council* untergeordnet sind. Diese fanden keine Thematisierung in den Interviews und werden hier weggelassen. Darunter befinden sich *Membership Boards* für Developer und Member, die das Bewerbungsverfahren durchführen. Mehr Infos finden sich bei Hill u. a. (2006) und auf <https://community.ubuntu.com/community-structure/>, letzter Aufruf 18.5.2017.

Ubuntu-Mitgliedern (*Community Council*) und den Ubuntu Developern (*Technical Board*) im Amt bestätigt. Mitglieder und Developer haben also eine kontrollierende Funktion. Des Weiteren gliedert sich die Struktur in *Teams*, die zu verschiedenen Themen arbeiten und von den beiden Gremien eingesetzt werden. Die *Local Community Teams* bilden in den verschiedenen Ländern lokale Infrastrukturen zur Koordination und stellen eine Brücke dar zwischen der zentralen Ubuntu-Organisation und den einzelnen lokalen Anwendergruppen (z.B. Ubuntu-Stammtische) in den verschiedenen Ländern.

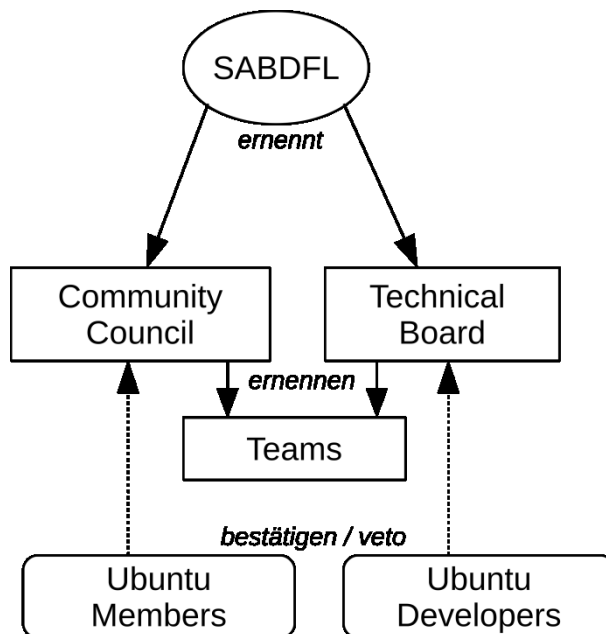


Abb. 8.2 Schematische Darstellung der Governance-Struktur in Ubuntu

Der „wohlwollende Diktator“ und Canonical

Der Gründer und Sponsor von Canonical hält sich offiziell als „selbst ernannter wohlwollender Diktator“ (SABDFL) weitgehend zurück aus den Entscheidungsprozessen und schaltet sich nur bei Schwierigkeiten in der Entscheidungsfindung ein, um dann eine notwendige Entscheidung zu ermöglichen. “In general he does get – he is the Self-Appointed Benevolent Dictator For Life, so he can, he *can* step in and make a final decision – and everyone is all right with that” (6,126).

Rein formell hat er wenig Entscheidungsbefugnisse, vielmehr werden Entscheidungen im Wesentlichen im *Technical Board* gefällt, bei dem Mark

Shuttleworth allerdings eine gewichtete Stimme hat, um Patt-Situationen und den damit verbundenen Stillstand zu vermeiden. Die Idee dahinter ist: Wenn es unter den Mitgliedern des Boards eine Pattsituation gibt, geht es mehr um die Durchführung einer Entscheidung als um die weitere Abwägung von Argumenten (vgl. Hill u. a. 2006: 305). Im Ubuntu-Wiki⁸⁸ wird das folgendermaßen ausgedrückt: “In many cases, there is no one ‘right’ answer, and what is needed is a decision more than a debate. The sabdfl acts to provide clear leadership on difficult issues, and set the pace for the project.”

Auch in Interviews wird Shuttleworth als zurückhaltend wahrgenommen (6,126), andere sind sich nicht sicher, welchen Einfluss Canonical tatsächlich ausübt (13,84). Jedoch wurden in den Interviews auch eine „strong ownership“ (11,120) bezüglich der Basiskomponente der Software („main packages“), eine mangelnde Nachvollziehbarkeit der Unternehmens-Strategie (11,112ff) und die generelle Einflussnahme (14,140f) kritisiert. “Obviously yes, it’s not a community driven distribution, right? It’s ultimately backed by Canonical, ... [who] *have* the power to just say that’s where we’re going, right?” (5,103).

Bezüglich formeller Rechte und Außenwahrnehmung gibt es hier offenbar eine Differenz. Shuttleworth äußert sich natürlich auch selbst auf seinem Blog bezüglich zu treffender Entscheidungen, aber formell entscheidet das Technical Board, in dem er nur mitentscheidet.

Das wird auch immer missverstanden: Wenn Mark Shuttleworth was sagt, heißt das erstmal im Grunde genommen gar nichts. Die Entscheidungsprozesse liegen erstmal beim Technical Board, wo er zwar auch Member ist und da auch mitreden darf, aber letztendlich entscheidet das Technical Board. (15,34)

Über diese Form der Stimmabgabe hinaus aber nominiert Shuttleworth sowohl die Mitglieder für das *Technical Board* als auch die Mitglieder für den *Community Council*, die aber von Mitgliedern der Community in ihrer Funktion bestätigt werden müssen (mehr dazu s. S. 161). Im Fall des *Technical Board* werden die vorgeschlagenen Mitglieder von den Entwickler*innen bestätigt (die für die Stimmberechtigung lediglich Schreibrechte im Repository haben müssen und folglich in der Mehrzahl unabhängig von Canonical

⁸⁸ <https://community.ubuntu.com/community-structure/governance/>, letzter Aufruf 9.5.2017

sind)⁸⁹, im Fall des *Community Council* sollen alle Ubuntu-Mitglieder die Vorgeschlagenen in ihrer Funktion bestätigen.⁹⁰

Über die formellen Einflussmöglichkeiten des SABDFL hinaus kann Canonical die Aktivitäten der dort angestellten Entwickler*innen dirigieren, was ein gewichtiger Einflussmechanismus ist, da in FLOSS-Projekten diejenigen Fakten schaffen können, die viel Zeit in die Entwicklung stecken. Ein Entwickler berichtet, dass manche Dinge auch zunächst in einem kleineren geschlossenen Kreis entwickelt und getestet werden. Erst im Laufe der Zeit wird dann das Feedback erweitert und hierbei zunächst ausgewählte Community-Mitglieder zum Testen eingeladen. Generell wird aber offenbar so viel wie möglich so offen wie möglich entwickelt (5,148ff).

So, you know, it's – we are trying to weight the pros and cons of keeping a certain thing under the wrap [...]. For example until we think it's ready for, you know, public consumption, for public critique as well, that's it. (5,158)

Einerseits wird dies begründet mit der Notwendigkeit, als gewinnorientiertes Unternehmen auf einem Markt mit Konkurrenten zu bestehen (5,147), andererseits ist das Feedback der Community in einer zu frühen Phase gegebenenfalls noch nicht unbedingt hilfreich: "... especially if we ask from a mailing list, whether we want, whether we switch to Unity or not, this would have been a flame war, and it would be – get us anywhere" (5,154).

Innerhalb des Unternehmens werden also durchaus auch Entscheidungen von der Unternehmensführung gegen den Willen der Entwickler*innen durchgesetzt (5,126). Offenbar erfolgte dies teils auch gegen die „partizipativ“ gefundenen Entscheidungen auf sogenannten Developer Summits (5,175). Diese Veranstaltungen, auf denen Entwickler*innen aus allen Ländern für ein Treffen und Kennenlernen im realen Leben zusammen kamen, um auch über künftige Entwicklungen zu diskutieren, gibt es aber in dieser Form nicht mehr (5,175). Wenngleich der Interviewte an dieser Stelle beklagt, dass die Partizipation an Entscheidungsfindungen auf den Summits nur Show war, bedauert er den Wegfall der Veranstaltung, da das Treffen in der Offline-Welt sehr positiv für die Entwickler-Gemeinschaft war.

Die Unternehmensstruktur hinter Ubuntu ermöglicht die Fokussierung von Ressourcen zur Realisierung dringend benötigter Komponenten und die zielgerichtete Verfolgung von Strategien. So gibt es bezahlte Mitarbeiter*in-

89 <https://wiki.ubuntu.com/TechnicalBoard>, letzter Aufruf 6.4.2017

90 <https://wiki.ubuntu.com/CommunityCouncil>, letzter Aufruf 6.4.2017

nen, die sich im Rahmen des Community Teams um die Community kümmern und die Werkzeuge und Plattformen der Mitglieder weiterentwickeln (15,65). Ebenfalls auffallend ist die Existenz eines Design Teams und eines Design Labs, das dezidiert User-Studien durchführt und User Experience als eigenständiges Expertise- und Innovationsfeld behandelt (5,114ff, 5,126). Die Fokussierung auf Design wiederum führt zu der stärkeren Integration weniger technikaffiner User (siehe 8.1.1) und das Community-Management hilft dabei, neue (beitragende) Mitglieder zu gewinnen (4,77).

Im Interview kam die strategische Führung von Canonical insbesondere zur Sprache in Bezug auf die Entscheidung, Ubuntu ab 2011 mit einer neuen Benutzeroberfläche namens Unity zu versehen, die eine eigene Entwicklung von Ubuntu ist und von anderen Distributionen nicht unterstützt wurde.⁹¹ Bei der Einführung gab es zunächst Widerstände von Nutzer*innen dagegen (15,147), vom altbewährten Standard GNOME abzuweichen.⁹² Dazu bemerkt ein Befragter, dass es in der Community häufig Widerstände gegen Neues gibt, hingegen Weiterentwicklungen aber beispielsweise aus einem Design-Standpunkt sinnvoll erscheinen (5,97). Es gibt aus dieser Perspektive also Entscheidungen, die zunächst nicht populär bei den Usern sein mögen, aber aus einem fachlichen oder strategischen Standpunkt sinnvoll und richtig erscheinen.

An diesem Beispiel wird auch sichtbar, dass manche Tools und Programme kurzerhand selbst entwickelt werden, wenn nach den gesetzten Kriterien ausreichende Alternativen fehlen (5,99f) – in diesem Fall ein intuitives Bedienkonzept. In solchen Fällen wird offenbar kurzfristig auf die Widerstände in der Community wenig Rücksicht genommen, um Innovationen durchzusetzen (5,143f).

91 Im Jahr 2017 wurde die Entwicklung von Unity wieder eingestellt, in der Zwischenzeit hat sich aber die von Ubuntu ursprünglich verwendete Oberfläche GNOME sehr stark weiterentwickelt und wird seit 2017 nun auch wieder von Ubuntu standardmäßig verwendet.

92 Weiterhin gibt es Varianten von Ubuntu, die andere Desktop-Oberflächen in das System einbetten, diese werden auch offiziell unterstützt. Varianten mit KDE (Kubuntu) und XFCE (Xubuntu) bestanden schon vor der Einführung von Unity, jedoch gab es nach der Einführung von Unity eine weitere Variante, die das ehemals als Standard verwendete GNOME verwendete: „Ubuntu Classic“. Diese Varianten werden „flavours“ genannt, siehe <https://www.ubuntu.com/about/about-ubuntu/flavours>, letzter Aufruf 5.4.2017).

Ein interessantes Beispiel für Dissenz zwischen Canonical und Community ist die Einführung einer integrierten Suchfunktion auf der Desktop-Oberfläche – die sogenannte *Dashboard-Suche* –, die beim Eingeben von Begriffen nicht nur nach Inhalten auf dem Rechner sucht, sondern die Suchbegriffe an ein Online-Versandhaus weiterleitet, um passende Produkte zum Kauf anzubieten. Dagegen regte sich in der Community starker Widerstand, der aber erst nach einiger Zeit berücksichtigt wurde.

Innerhalb der französischen Ubuntu Community führte dies sogar dazu, dass das neue Release in Frankreich keine reine Übersetzung war, sondern ein echter „Fork“, also eine veränderte Distribution: Das Französische Community Team baute in der französischen Variante von Ubuntu einen Schalter ein, mit dem die kritisierte Funktion deaktiviert werden konnte. Die französische Version wurde anschließend als veränderte Variante veröffentlicht: „... it was the first time the french version of ubuntu wasn't just a translation“ (14,204).

In der folgenden Ubuntu-Version wurde dann auch eine derartige Opt-out-Lösung in die originale Ubuntu-Version eingebaut. Dieser Änderung seitens Canonical ging also reger Protest in der Community voraus und sogar eine Veränderung des Codes der französischen Variante, die eigentlich nur als Übersetzung gedacht ist. Hier spielte also die Möglichkeit des Forks, der Anpassung und Veränderung des Systems eine politische Rolle für die konkrete Gestaltung.

Die beiden Beispiele Unity und Dashboard-Suche zeigen, wie Canonical zunächst ohne Rücksicht auf Feedback aus der Community Änderungen einführt, die seitens Canonical als sinnvoll erscheinen. Die rege Kritik der Community kann das Unternehmen aber durchaus zum Einlenken bringen. Schließlich besteht bei konstanter Unzufriedenheit die Gefahr, dass User und Developer abwandern und beispielsweise ein Fork entsteht wie bei der französischen Variante. „I think there are different levels of things, it's – when you are speaking about is it beautiful, I think they are just deciding on their own, but when you are speaking about privacy and security, they are able to hear what we have to say“ (14,147).

Der Einfluss von Canonical umfasst also die Weisungsbefugnis über bezahlte Entwickler*innen und über den SABFDL Mitbestimmungsrechte bei der Zusammensetzung der zentralen Gremien sowie eine gewichtete Stimme bei technischen Fragen. Die Entscheidungsmacht wurde in der Vergangenheit auch verwendet, um bestimmte Entwicklungen voranzutreiben, in der Auseinandersetzung mit der Community konnte aber in Bezug auf den

Schutz der Userdaten bei der Dashboard-Suche ein Einlenken beobachtet werden.

Technical Board und Community Council

Im Wesentlichen gibt es zwei wichtige Gremien, die jeweils für zwei verschiedene Bereiche zuständig sind: Das *Technical Board* entscheidet über technische Fragen und der *Community Council* verhandelt Community-bezogene Themen. Zusammen bilden diese die wichtigsten Governance-Organen (vgl. Hill u. a. 2006: 299ff.). Dem *Community Council* sind die *Membership Boards* untergeordnet, die aufgeteilt nach lokalen Regionen und mit einem separaten Board für Developer-Anträge über Mitgliedsanträge abstimmen. Diese wurden auch explizit in den Interviews genannt. Weitere Gremien finden sich auf der Community-Website⁹³, sie fanden aber keine Erwähnung in den Interviews, weshalb ich sie in der folgenden Betrachtung vernachlässige.

Technical Board Neben der Klärung von Schwierigkeiten bei der Koordination zwischen verschiedenen Entwickler*innen ist das *Technical Board* verantwortlich für richtungsweisende Entscheidungen. Darunter fallen mitunter Entscheidungen bezüglich der unterstützten Software, des Installationsprozesses, der verwendeten Tools sowie der Abhängigkeiten der Softwarebibliotheken und anderer technischer Angelegenheiten, die „technical supervision“ erforderlich machen.⁹⁴ “If someone needs to step in and make a final decision, there’s a tech board” (6,88).

Zentral sind dabei die Bestimmung der nächsten Entwicklungsziele (*Ubuntu Release Feature Goals*), da hier die Prioritäten für die zukünftige Gestaltung der Software definiert werden, sowie die Definition der Standard-Konfiguration, also welche Programme auf welche Weise vorausgewählt werden und die übliche Erscheinung des Produkts prägen (*Ubuntu Package Selection*). Das Gremium besteht aus sechs vom SABDFL Mark Shuttleworth ernannten Mitgliedern. Diese werden von ihm vorgeschlagen und durch eine Wahl aller Entwickler*innen im Amt bestätigt oder abgelehnt, hierbei sind alle Entwickler stimmberechtigt, die Schreibrechte an einem Repository haben.

93 <https://community.ubuntu.com/community-structure/>, letzter Aufruf 6.4.2017

94 <https://wiki.ubuntu.com/TechnicalBoard>, letzter Aufruf 6.4.2017

Anliegen aus der Community können an das wöchentlich tagende Gremium (in einem Internet Relay Chat) über eine spezielle Mailingliste hergetragen werden. Diese nehmen die Themen dann auf die Agenda auf, falls es Diskussionsbedarf gibt. Bei Abstimmungen innerhalb des Gremiums gilt die einfache Mehrheit, bei Stimmgleichheit entscheidet die Stimme des SABDFL (gewichtete Stimme).⁹⁵

Community Council Eine Besonderheit im Fall Ubuntu ist die Existenz eines Gremiums, das sich mit Angelegenheiten beschäftigt, die nicht technischer Art sind, sondern die Community selbst betreffen. Diese Angelegenheiten werden vom *Community Council* verhandelt. Er ist auch mit der Pflege der “social structures, venues, and processes of the project” (ebd.: 299) betraut. Darunter fällt beispielsweise der *Code of Conduct*, also des Dokuments, das den sozialen Umgang der Mitglieder untereinander regelt. Außerdem werden auch zwischenmenschliche Konflikte oder Spannungen, die sich in der Interaktion ergeben (z.B. auf der Mailingliste), angesprochen und Lösungen dafür gesucht (15,90). Das Gremium verwaltet alle “community related structures and processes” (ebd.) und stellt Teams für bestimmte Aufgaben zusammen und löst diese gegebenenfalls auch wieder auf. Schließlich ist die Bearbeitung von Mitgliedsanträgen an das *Membership Board* ausgelagert, das dem *Community Council* untersteht. Der *Community Council* kann für bestimmte Entscheidungen auch ein Wahlverfahren einberufen, eine sogenannte Resolution, und somit die *Ubuntu Members* über eine anstehende Entscheidung abstimmen lassen.

Die Abwägung, ob eine Angelegenheit eher dem *Technical Board* oder dem *Community Council* zugehörig ist, wird ad hoc entschieden und – soweit es nicht ganz offensichtlich ist – gegebenenfalls nach Rücksprache mit anderen im Chat (Internet Relay Chat, kurz IRC) einem Gremium zugewiesen: „Dann fragt man einfach mal so rum, und dann wird nach Stimmungsbild gefragt und gegebenenfalls über die Mailingliste kommuniziert“ (15,83).

Interessant ist hierbei, dass sowohl der SABDFL Vorschlagsrechte für die Mitglieder in den Gremien besitzt als auch die offiziellen Mitglieder (so auch die Nicht-Entwickler*innen) die Beisitzenden im *Community Council* bestätigen müssen. Hier hat also neben dem „Chef“ auch jedes einzelne Mitglied verankerte Mitbestimmungsrechte – zumindest im nicht-technischen Community-Bereich.

95 Vgl. <https://wiki.ubuntu.com/TechnicalBoardAgenda>, letzter Aufruf 26.3.2018.

Neben dem *Community Council* gibt es außerdem sogenannte *Local Communities* als kleinere Organisationseinheiten. Diese organisieren auf Landesebene die Kommunikation der gesamten Community und repräsentieren die User ihrer örtlichen Zuständigkeit. Die Organisation dieser „LoCos“ ist sehr heterogen, in manchen werden Vorstände gewählt, in anderen macht mit, wer Lust hat (15,26f).

Teams Neben den oben erwähnten *Community Team* und dem *Design Team* gibt es viele weitere Teams zu einzelnen Bereichen von Ubuntu. Sie fanden in den Interviews keine besondere Erwähnung, aber laut der Beschreibung auf der Website können sich an den regelmäßigen Online-Sitzungen (über IRC) interessierte User an Diskussionen und Entscheidungen beteiligen. Die Teams werden formell vom *Community Council* koordiniert, der auf Anfrage auch neu initiierte Teams in die Struktur einbindet.⁹⁶

Code und Diskurs – Feedback und Mitgestaltung durch die Community

Jenseits der formalen Strukturen werden kleinere Fragen und Entscheidungen auch ad hoc im IRC Channel (6,III) oder auf den regelmäßigen Online-Meetings diskutiert, die innerhalb thematisch untergliederter Teams stattfinden. Die Diskussion ist Open-Source-typisch dabei in der Regel zielorientiert und pragmatisch, und im Zweifelsfall treffen die erfahreneren Mitglieder die Entscheidungen – ganz meritokratisch.

I can't think a single time that we discussed things, where two people were saying no I want this, no I want that. That doesn't seem to happen. Basically it's more of a meritocracy, so in general when two people disagree, there will be someone who's considered more into it with the code, and will just, they'll just make a decision. (6,II9)

Schließlich herrscht das Credo der technisch überlegenen Lösung, und wer partout nicht einverstanden ist, kann sich jederzeit eine eigene Variante, einen eigenen Fork abspalten. “We don't have to all agree on everything, right? That's were the – the good thing about [how] open source goes is that, you know, you're open to do your own thing, not everybody has to agree” (5,105).

Neben der aktiven Gestaltung des Codes haben Entwickler*innen also die Möglichkeit der Mitbestimmung bezüglich der Mitglieder des *Technical Board* und können auch als Mitglied des Gremiums nominiert werden. Des Weiteren haben bei Ubuntu – anders als bei den anderen betrachteten Fällen

96 Siehe <https://wiki.ubuntu.com/Teams>, letzter Aufruf 5.4.2018.

– auch explizit Nicht-Entwickler*innen die Möglichkeit, einerseits relativ niederschwellig Feedback durch einfache Bug Reports und Ähnliches zu geben (s. Abschnitt 8.3.1), andererseits haben sie als formale Mitglieder Einfluss auf die Besetzung des *Community Council* und das Recht, hier bei Abstimmungen teilzunehmen. Jedoch bleibt die Mitbestimmung der Community bezüglich der Besetzung der beiden zentralen Entscheidungsgremien relativ schwach, da deren Mitglieder durch den „Diktator“ (SABDFL) nominiert werden.

8.2.2 Debian – Demokratie der Entwickler*innen

In Debian gibt es starke Entscheidungsorgane, die auch demokratische Entscheidungen zu Wege bringen können. Auf der Makro-Ebene liegt die maßgebliche Entscheidungsmacht bei den anerkannten *Debian Developers*, sie wählen den *Debian Project Leader* und können ein Abstimmungsverfahren einleiten, die *General Resolution*, bei der über ein Wahlverfahren demokratisch Entscheidungen getroffen werden. Dezidiert „technische“ Entscheidungen werden aber von einem *Technical Committee* entschieden.

Auf der Mikro- und Meso-Ebene ist die Organisationsform eher dezentralisiert gesteuert von kleineren Arbeitsgruppen, die sich untereinander koordinieren und Entscheidungen fällen. Hier ist jede*r für den Bereich zuständig, an der sie oder er konstruktiv mitarbeitet.

It's decentral, I mean, if you don't have a *big* decision like changing the init system or something like that, it's basically free for all ... I don't know if that's an anarchic system or something like that, but each person, or contributor or each group of contributors that are working on a project or a group of packages, are taking their own decisions as long as you are not annoying the people next there, I mean, you can do whatever you want. (11,105)

Mit Verweis auf diese dezentrale Struktur bemerken einige Befragte auch, dass es schwer fällt, generelle Aussagen über die Arbeitsstrukturen zu treffen (4,76ff), da die Community sehr heterogen mit unterschiedlichen Leuten und verschiedenen Interessen ist (2,171). Wir wollen hier dennoch eine Analyse wagen.

deren Anerkennung als *Debian Developer* können sie an den Entscheidungen der Community mit Stimmrecht teilnehmen (vgl. dazu auch die Problematisierung in Lazaro 2008: 115). Der Anteil der *Debian Developer* ohne Uploadrechte liegt allerdings bei lediglich knapp 3%⁹⁷ und fand auch in den Interviews keinerlei Erwähnung. Diese spezielle Rolle hat also noch keine große Relevanz in der Community. Allerdings ging der Einrichtung dieser speziellen Rolle eine Diskussion voraus und wurde über eine General Resolution mit der Mehrheit des nötigen Quorums der Entwickler*innen befürwortet.⁹⁸

Koordination der Teams und Maintainer Bezüglich der Koordination der Entwickler*innen fällt auf, dass im Grunde jeder Maintainer zunächst für die ihm zugewiesenen Pakete verantwortlich ist (6,88, 11,105f, 2,150). Hier haben die Entwickler*innen große Freiheit, *woran* gearbeitet wird – “I can work on whatever I want” (2,34). Auch die Dokumentation und Übersetzung ist in der Hand der Maintainer (2,54).

Darüber hinaus gibt es aber Arbeitsgruppen, die sich abstimmen, insbesondere bei Paketen, die aufgrund ihrer logischen Struktur zusammenhängen (11,105f). Die Maintainer entscheiden dann in gegenseitiger Abstimmung, basierend auf ihrer Erfahrung (Meritokratie) und in Rücksprache mit anderen Entwickler*innen. (2,150) Ein User beschreibt diese Form der Kooperation als „interaktive“ und „natürliche“ Bottom-up-Entscheidungsfindung, weil so die Maintainer mitreden, um die es bei einer spezifischen Entscheidung geht, anstelle von hierarchischen Gruppen, die Entscheidungen von oben fällen, und weil nicht eine kleine Gruppe von Entwickler*innen das große Ganze kontrolliert (13,84).

With Debian, it's a lot more kind of bottom up-ish – I think it's a lot more interactive as far as the organisation – it's not like a little group of people is controlling the entire direction of debian, there are like, package maintainers, that get a little bit of say, you know people are on certain projects they get to say – and it kind of evolves more naturally. Whereas in Ubuntu it is a little more top-down. (13,84)

97 Insgesamt gibt es 28 non-uploading DD (https://nm.debian.org/public/people/dd_nu) und 974 uploading DD (https://nm.debian.org/public/people/dd_u, letzter Aufruf 28.3.2018).

98 Vgl. die Ankündigung der „Guidelines for applying as non-uploading DD“ (<https://lists.debian.org/debian-devel-announce/2010/11/msg00000.html>) und die Auszählung der Stimmen (https://www.debian.org/vote/2010/vote_002; beides zuletzt aufgerufen 7.4.2017).

Ein User nennt die Arbeitsweise eine „Do-ocracy“ (11,139), in der jede*r etwas tun kann, solange es niemand anderen stört – was in gewisser Weise auf alle Open-Source-Projekte zutrifft, aber hier in Form der Entwickler-Struktur institutionell unterstützt wird. Also diejenigen, die etwas machen, entscheiden auch, wie sie es machen. Mit anderen Worten: “ultimately what’s matter is in the code and what ships so – it is the one who is doing the work who decides, and in debian it is really that way” (2,50).

Dabei wirkt die Möglichkeit, Änderungen jederzeit wieder rückgängig machen zu können, auch entlastend für die Maintainer (2,151f): “the good thing with IT, what i love with computer science, the one of the – only discipline where you can push something, break it, and roll back the change“ (2,150).

Das Technical Committee Im größeren Zusammenhang kann die relative Autonomie der einzelnen Pakete oder Paketgruppen aber für das gemeinsame Zusammenspiel von Programmpaketen zu Schwierigkeiten führen (11,139). Daher gibt es ein übergeordnetes Koordinationsinstrument: das *Technical Committee*. Bei größeren technischen Entscheidungen, die eine Koordination notwendig machen, oder wenn es überschneidende Interessen zwischen Maintainern gibt – und diese sich nicht einigen können oder wollen –, kann das Technical Committee angerufen werden. Das Technical Committee kann dann die Entscheidungen einzelner Maintainer „überschreiben“, also über die individuellen Ansichten von einzelnen Mitgliedern hinweg Entscheidungen durchsetzen. Ein simples Beispiel für eine derartige Entscheidung ist, wenn zwei unterschiedliche Programmpakete denselben Namen für die aufrufende Datei nutzen, ein ebenso trivialer wie problematischer Konflikt (2,314). Die Durchsetzung einer Entscheidung durch das Technical Committee ist jedoch aufgrund der möglichen Spannungen ein weniger favorisierter Weg, kommt aber in der Praxis auch nicht häufig vor (2,320f).

Das Technical Committee kann von einem *Debian Developer* angerufen werden, wenn es in einem wichtigen Punkt Uneinigkeit gibt. Nach gründlichem Erwägen und Austauschen von Argumenten wird dann schließlich abgestimmt. Das Technical Committee hat eine*n Vorsitzende*n, der oder die bei gleicher Stimmenanzahl eine gewichtete Stimme hat und somit die Entscheidung treffen kann.

Die prominenteste und härteste Kontroverse in jüngster Zeit war die Diskussion über den Wechsel des Startprozesses des Systems – eine sehr grundlegende Entscheidung, da der altbewährte Prozess seit Jahrzehnten etabliert war und die Systeminitialisierung eine sehr fundamentale Angelegenheit

darstellt. In diesem Fall gab es in Debian eine jahrelange, hitzige Debatte, die schließlich durch eine Entscheidung des Technical Committee zu einem Ende gebracht wurde. Hier zeigt sich das Konfliktpotenzial auch „technischer“ Entscheidungen sowie das entschärfende Potenzial einer Entscheidungsstruktur (12,32). Jedoch bedarf es gerade in solchen zentralen Fragen einer sorgfältigen Abwägung, welcher Sachverhalt genau abgestimmt werden soll⁹⁹ – ob beispielsweise der alte Prozess vollständig ersetzt werden soll oder ein weiterer optional angeboten werden soll. Ebenso ist es diskutabel, ob eine Entscheidung im *Technical Committee* entschieden wird oder ob eine *General Resolution* durchgeführt werden soll. Die Definition der technischen Entscheidung“ sowie die Abgrenzung zwischen „technischer Entscheidung“ und „politischer Entscheidung“ sind also in gewissem Sinne selbst politische Fragen.

Wahlverfahren: Die „General Resolution“ Ein bemerkenswertes Instrument zur Entscheidungsfindung ist die sogenannte *General Resolution* – ein Wahlverfahren, bei dem alle *Debian Developer* eine Stimme haben, das aber vorrangig für *politische* Entscheidungen vorgesehen ist, solche also, die nicht eindeutig mit allein technischen Kriterien bewertet werden können (2,325). Ein Beispiel für eine politische Entscheidung ist der oben erwähnte Beschluss, die Mitgliedschaft in Form der Rolle *Debian Developer* auszuweiten auf Teilnehmer der Community, die nicht am Code der Software-Pakete mitarbeiten und die Rolle *Debian Developer non-uploading* zu installieren (s. S. 165). Ein anderes Beispiel wäre eine Änderung des *Code of Conduct* (2,334). In diesem und ähnlichen Fällen wird von einer kleineren Gruppe ein Änderungsvorschlag erarbeitet, der dann zur Wahl gestellt wird. Für den Abstimmungsprozess gibt es eine separate Mailingliste „debian-vote“.

Ein Wahlverfahren kann – wenn anderweitig kein Konsens erreicht wird – Diskussionen abkürzen und eine Entscheidung auf den Weg bringen, was bei anderen Projekten teils schwerer fällt, wie ein Developer über eine andere Community (Mozilla) berichtet (2,350). Beim Wahlverfahren der *General Resolution* hat jede*r einzelne Entwickler*in (über 1000 an der Zahl)¹⁰⁰ eine Stimme. Außerdem werden vom Debian Project Leader Diskussions- und Abstimmungszeiträume festgelegt. Dadurch kann sich die Entscheidungsfindung über mehrere Wochen hinziehen und wird daher als sehr aufwendig

⁹⁹ Vgl. die initiale Anrufung des TC und die damit verbundene Diskussion, <https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=727708>, letzter Aufruf 28.3.2018.

¹⁰⁰ Stand 28.3.2018: 1002, siehe Fußnote 97.

wahrgenommen. Eine derart getroffene Entscheidung der gesamten *Debian Developer* wird gegenüber andere Entscheidungen des DPL oder des *Technical Committee* priorisiert; deren Entscheidungen werden durch die *General Resolution* „überschrieben“ (12,20).

Dies wird unterschiedlich bewertet, einerseits ist ein Wahlverfahren geeignet, um die Akzeptanz von Entscheidungen zu stärken, andererseits sind die Entscheidungsverfahren mit viel Aufwand und Energie durch lange, teils langwierige Diskussionen verbunden (11,47).

Debian Project Leader und Delegations

An der Spitze des Debian-Projektes steht der *Debian Project Leader* (DPL), der aber im Gegensatz zum selbsternannten Diktator bei Ubuntu jährlich gewählt wird, wobei jeder *Debian Developer* mitwählen darf. Vor der Abstimmung kann sich jeder *Debian Developer* zur Wahl aufstellen, seine Visionen und Ziele darstellen und den anderen Bewerber*innen Fragen stellen; so entsteht eine Art Wahlkampf, nach dessen Ende die Abstimmung durchgeführt wird. In erster Linie hat der DPL eine repräsentative Funktion sowohl nach außen zur Vertretung des Projekts als auch nach innen zur Unterstützung und Gestaltung des Projekts.¹⁰¹ Außerdem werden vom DPL Entscheidungen gefällt, die dringend sind, für die niemand sonst zuständig ist – ferner, wie das verfügbare Budget verwendet wird (2,313f).

Darüber hinaus gibt es eine weitere Organisationsstruktur, die sogenannten *Delegations*. Diese haben definierte Aufgabenbereiche und werden vom Debian Project Leader eingerichtet und personell besetzt (12,13).¹⁰² Ein solcher Aufgabenbereich ist zum Beispiel die Begrüßung und Einführung neuer Mitglieder oder die Entscheidung, welche Software in die Distribution aufgenommen wird (12,13). Die Entscheidungen der *Delegations* sind bindend und können vom DPL nicht zurückgenommen werden, jedoch kann der DPL die Gruppe einer Delegation auflösen – was sich jedoch nicht auf deren getroffene Entscheidungen auswirkt.

Damit kommt dem DPL ein gewisser Einfluss zu, der aber dadurch begrenzt ist, dass weiterhin alle Tätigkeit innerhalb Debian auf Freiwilligkeit basiert und daher auch die Delegierten ein eigenes Interesse haben müssen, die ihnen zugewiesenen Aufgaben zu erfüllen.

101 Siehe <https://www.debian.org/devel/leader>, letzter Aufruf 10.4.2017.

102 Für ein Beispiel für die Einsetzung einer Delegation vgl. <https://lists.debian.org/debian-devel-announce/2015/03/msg00010.html>, letzter Aufruf 12.4.2017.

Die User

Ein User kritisiert, dass die Entscheidungen allein von den Entwickler*innen getroffen werden, und bemängelt die fehlende Einbindung von Usern. Gerade Power-User und Administrator*innen sind sehr stark von grundsätzlichen Entscheidungen betroffen. Wenn beispielsweise der Initialisierungsprozess des Systems (s. S. 167) geändert wird, bedeutet das für die Art von Usern, die eine große Anzahl von Computern betreuen, unter Umständen viel Arbeit (10,180ff). Der User des Systems spielt also in den Strukturen bislang eine untergeordnete Rolle (12,7). Eine maßvolle Veränderung deutet sich hierbei aber in der Berücksichtigung von nicht-entwickelnden (non-uploading) Beitragenden für das Bewerbungsverfahren an (s. S. 165).

8.2.3 Arch – informelle Oligarchie

In Arch bilden die *Arch Developer* die zentrale Governance-Ebene. Sie spielen eine tragende Rolle für die Führung der Community, allerdings ist dies ein relativ kleiner Kreis von weniger als 40 Entwickler*innen (Stand 10.4.2017)¹⁰³ und somit ist hier eine informelle Kommunikation relativ leicht, da die Entwickler*innen untereinander bekannt sind.

I have to say we are very cohesive, we stick together ... Because every decision we take, for example when I want to ... change something, I write to the others and say I want to do this. I never saw someone complain, say no you should not, everytime I see "OK" – in the worst case I see "do what you want". This means that we share the same idea. (1,117f)

Darüber hinaus gibt es 50 Trusted User (Stand 10.4.2017),¹⁰⁴ die zwar nominal als User bezeichnet werden, aber ebenfalls Entwickler-Aufgaben erledigen.

„Half-gods working to improve Arch“: Developers Die ironische Bezeichnung als Halb-Götter findet sich in der „Arch-Terminologie“ auf der Website des Projekts.¹⁰⁵ Wengleich ironisch überhöht, spiegelt die Bezeichnung ein elitäres Grundverständnis der Arch Community wider. Die *Arch Developer* treffen recht unbürokratisch die wesentlichen Entscheidungen für

103 <https://www.archlinux.org/people/developers/>, letzter Aufruf 10.4.2017.

104 https://wiki.archlinux.org/index.php/Trusted_Users#Active_Trusted_Users, letzter Aufruf 10.4.2017

105 https://wiki.archlinux.org/index.php/arch_terminology#Developer, letzter Aufruf 10.4.2017

das Projekt. In Bezug auf die oben diskutierte Entscheidung des Wechsels des Startprozesses – eine Entscheidung, die durch die zunehmende Etablierung des neuen Startprozesses „systemd“ und den damit verbundenen Abhängigkeiten in anderen Programmen jede Distribution zu fällen hatte – zeigt sich hier ein deutlicher Unterschied in der Entscheidungsfindung zu der langwierigen Diskussion bei Debian (vgl. S. 167). Während in der Entwicklerliste der Entscheidungsprozess sich von Anfang an konstruktiv darum dreht, wie ein nahtloser Übergang hin zum neuen Startprozess erreicht werden kann¹⁰⁶ – und nicht darüber, ob der Wechsel *überhaupt* geschehen soll –, kündigt ein User auf der Userliste eine Meinungsumfrage zum geplanten Wechsel an. Darauf reagiert ein Entwickler wortkarg: “With or without this poll, we are continuing with our plan” – und stellt damit unmissverständlich klar, dass eine User-Umfrage keinerlei Einfluss auf die Entscheidungsfindung der *Arch Developer* hat.

- > Yesterday I read on Phoronix that Arch devs are planning to switch to
 - > SystemD [...] I have created an online poll to
 - > determine the will of the community [...]
 - > Please vote and spread!
- with or without this poll, we are continuing with our plan.

Die Entwickler¹⁰⁷ sind also eine relativ kleine homogene Gruppe von engagierten Entwicklern, die für „das Ganze“ entscheiden (1,126; 8,279). Darüber hinaus pflegen und kontrollieren sie die wichtigsten Software-Pakete von Arch, das sogenannte *Core Repository*.

Trotz der informellen Struktur gibt es klare Kriterien wie höchste Aktualität („bleeding edge“), Orientierung am Upstream und Minimalismus („Keep It Simple, Stupid“). Arch bildet damit eine Distribution, die möglichst direkt und unverändert die Fülle der Open-Source-Projekte zur Verfügung stellt. Durch die pure Anbindung an die Upstream-Projekte muss wenig distributionspezifisch angepasst werden und es können ohne viel Aufwand stets die aktuellsten Versionen mit einbezogen werden.

106 Siehe <https://lists.archlinux.org/pipermail/arch-dev-public/2012-August/023389.html> und folgende, letzter Aufruf 28.3.2018.

107 Die *Arch Developer* sind zum Zeitpunkt des Verfassens dieses Manuskriptes tatsächlich alle männlich.

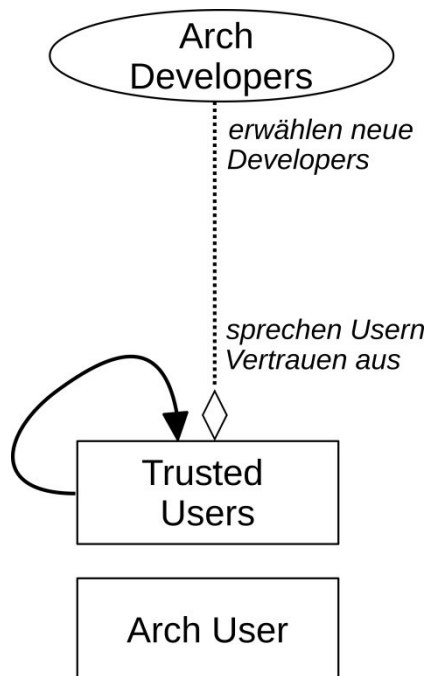


Abb. 8.4 Schematische Darstellung der Governance-Struktur in Arch

In anderen Distributionen sind auch deutlich mehr Entscheidungen nötig, gerade in Bezug auf die distributionsspezifische Konfiguration der Upstream-Software. Vor allem aber die Pflege verschiedener Distributionskanäle wie Debians Unterteilung in Oldstable, Stable, Testing, Unstable und Experimental bedeutet einen hohen Aufwand, da auch für ältere unterstützte Versionen Sicherheits-Updates in den Code eingepflegt werden müssen. Arch hingegen konzentriert sich auf einen Distributionskanal, der sehr nahe am aktuellen Upstream ist, lediglich mit einer Vorstufe, dem *Testing Repository*¹⁰⁸. Es werden aber keine älteren Versionen weiter parallel gepflegt. Der minimalistische Ansatz macht es möglich, die Distribution mit einem relativ kleinen Kreis von Entwicklern zu pflegen und erleichtert es ihnen schließlich, sich auf einen gemeinsamen Kurs zu verständigen.

Unter Entwicklern gibt's eigentlich ne flache Hierarchie. Es gibt mehrere Leute, die haben einen vollen Zugriff auf alle Systeme, und es funktioniert trotzdem alles. Also, dass manches sich irgendwie gut ergänzt – also es gibt Gebiete, da sind dann drei bis vier Leute Experten, die machen das unter sich aus,

108 “[T]esting contains packages that are candidates for the core or extra repositories.“, https://wiki.archlinux.org/index.php/Official_repositories, letzter Aufruf 11.4.2017

und die anderen stört das nicht weiter – ohne dass man jetzt festlegt, „der gibt den Ton an, wir gehen jetzt in ne Marschrichtung, das ist unser Konzept“. Das hängt vielleicht ein bisschen damit zusammen, dass wir ein minimales System anbieten und minimale Grundsätze haben, dass wir nicht irgendwie ne große Vision haben oder ne große Sache. Sondern wir können einfach – das Minimal-Set steht fest, da ist sich auch jeder einig, und da kann man viel machen. Also wir haben jetzt nicht sowas [wie] „wir müssen unbedingt auf mobile [Umgebungen]“ oder so, das wäre eigentlich gar kein Thema, über das man sprechen müsste. (8,271ff)

Schließlich kommt der minimalistische Ansatz auch Software-Entwicklern aus den Upstream-Projekten entgegen. Einerseits kann hier die Software leicht in der originären Konfiguration bezogen werden, was für die Mitentwicklung wichtig ist. Ebenso entstehen relativ wenige Fehler auf der Distributionsebene durch Anpassung und Konfiguration, das macht generell die Zuarbeit für Upstream-Projekte einfach und wahrscheinlich, da auftretende Fehler häufig direkt dem Upstream gemeldet werden können und so auch die Fehlerbehebung aus dem Upstream sehr schnell wieder zurück in die Distribution fließt.¹⁰⁹

Für den Status eines *Arch Developer* gibt es – anders als bei den anderen Fällen – kein formales Bewerbungsverfahren (8,281). Stattdessen werden bei Bedarf Mitglieder der Community rekrutiert, die sich üblicherweise schon als Trusted User bewährt haben (8,299; 1,130) durch die Arbeit an den von den Trusted User betreuten Programmpaketen oder beispielsweise durch Mitarbeit an der Website oder an Arch-spezifischen Skripten und Software-Tools (1,135). Die Institution der Trusted User dient somit auch der „Nachwuchsförderung“ (8,332). In wenigen Fällen werden auch Entwickler*innen aus bestimmten Upstream-Projekten angesprochen, ob sie *Arch Developer* werden wollen, wenn es hier einen besonderen Bedarf gibt (1,131).

Im Wesentlichen betrachten Entwickler die Unterschiede zwischen Trusted Usern und *Arch Developer* als wenig bedeutend, abgesehen davon, dass eben *Arch Developer* die offiziellen Repositories (insbesondere *Core* und *Extra*)¹¹⁰ betreuen und Trusted User das *Arch User Repository* und das

109 Mehr zur Bedeutung von Upstream und Downstream in Distributionen siehe S. 106.

110 *Core* enthält die wichtigsten Pakete für den Systemstart, die Verbindung mit dem Internet, das Bauen von Software-Paketen, Dateisysteme und die System-Pflege und die dazugehörigen Programme („Abhängigkeiten“); *Extra* enthält alle Pakete, die nicht in *Core* passen (s. https://wiki.archlinux.org/index.php/Official_repositories, letzter Aufruf 11.4.2017).

Community-Repository (I,126) pflegen. Technisch betrachtet, müssen Trusted User ebenso vertrauenswürdig sein, da sie mit ihren Schreibrechten auch sorgfältig arbeiten müssen, um das System intakt zu halten (8,307)

Trusted User und User Neben den *Arch Developern*, die neben der Verwaltung der zentralen Software-Repositories auch die richtungsweisen Entscheidungen in Arch vorgeben, gibt es als formale Rolle die *Trusted User*. Diese haben ebenfalls Schreibrechte, verwalten aber das *Community*-Repository und supervidieren das *Arch User Repository* (AUR) (8,279).¹¹¹

Das AUR ist zwar offen für alle User zum Bereitstellen von Software, wird aber von den Trusted Usern beaufsichtigt. Als übergeordnete Instanz legen die Trusted User auch fest, welche Pakete im AUR zusammengelegt werden sollen, können Pakete löschen oder verwaiste Pakete anderen Usern zuweisen, damit diese sie pflegen können. Eine detaillierte Qualitätsüberprüfung über die im AUR zur Verfügung gestellten Pakete können die Trusted User aber nicht leisten, da dies schlicht zu viel ist (8,325ff). Hier sind also die User selbst in der Pflicht, die bereitgestellten Pakete gegebenenfalls zu überprüfen. Mit den Aufgaben der Trusted User geht eine hohe Verantwortung einher, da die Änderungen am *Community*-Repository auch das ganze System betreffen können, daher müssen sich User erst im AUR durch gute Arbeit an den von ihnen bereitgestellten Paketen bewähren, bevor sie sich als Trusted User bewerben können (8,280). Anders als für den Status der *Arch Developer* ist eine Bewerbung für den Status als Trusted User aber möglich. Dafür benötigt der oder die Bewerber*in – ähnlich wie bei den anderen betrachteten Fällen – eine*n Unterstützer*in, der oder die bereits den Status eines Trusted User hat. Die Bewerbung findet zunächst informell statt, indem ein User einen Trusted User anspricht, ob er oder sie ihn für die Bewerbung unterstützt. Ist ein*e Unterstützer*in gefunden, wird die Bewerbung in der Mailingliste der Trusted User diskutiert und darüber abgestimmt (I,148ff). Vereinzelt kommt es vor, dass Bewerber*innen abgelehnt werden (I,143). Erreicht der oder die Bewerber*in das nötige Quorum (I,156ff), müssen drei *Arch Developer* ihr Vertrauen aussprechen, bevor er oder sie Schreibrechte im Repository bekommt (8,281). Dies geschieht in der Praxis durch kryptografische Signaturverfahren.

111 Siehe auch https://wiki.archlinux.org/index.php/Trusted_Users, mehr zu den offiziellen Repositories findet sich unter https://wiki.archlinux.org/index.php/Official_repositories, letzte Aufrufe 11.4.2017.

Durch die Empfehlung eines Trusted Users finden sich für gewöhnlich weitere Trusted User, die dem oder der Anwarter*in durch die Signierung ihres Schlüssels dem angehenden neuen Trusted User ihr Vertrauen aussprechen (8,288ff).

Arch User Die „normalen User“ haben insofern ähnliche Aufgaben wie die Entwickler*innen, als sie aufgerufen sind, Software für die anderen User im AUR bereitzustellen.

Somit fallen sie auch unter die Definition der *Maintainer*¹¹², die drei verschiedene Stufen von Maintainern bezeichnet:

- “A core Arch Linux developer who maintains a software package in one of the official repositories (core, extra, or testing).
- A Trusted User of the community who maintains software packages in the unsupported/unofficial community repository.
- A normal user who maintains a PKGBUILD and local source files in the AUR.”

Über das AUR kann also jeder User die Distribution über die Bereitstellung von Software-Paketen mitgestalten und kann über die dort bereitgestellten Pakete abstimmen und somit dazu beitragen, dass beliebte Pakete in das *Community*-Repository übernommen werden. Darüber hinaus haben natürlich die User auch die Möglichkeit der Gestaltung der Community durch das Beitragen von Anleitungen, Bug Reports und Übersetzungen (vgl. Abschnitt 8.3.3).

Nichtsdestrotz werden die grundsätzlichen Entscheidungen für das Projekt als Ganzes von der Elite der *Arch Developer* getroffen und es besteht kein formelles Verfahren, in diese Gruppe aufzusteigen. Möglich ist lediglich eine Bewerbung für den Status des Trusted User, aus dem man zum *Arch Developer* erwählt werden *kann*, aber nicht notwendiger Weise muss: „Wir haben viele, ... ehemalige Trusted User, die dann Entwickler geworden sind, das ist – ja, ich würd’ jetzt nicht sagen ne Laufbahn, weil man muss nicht Entwickler werden, das ist eher so ne Bedarfssache auch“ (8,299).

8.2.4 Vergleich der Variable „Strukturen und Verfahren“

Wie an den schematischen Darstellungen¹¹³ anschaulich wird, sind die organisatorischen Strukturen in den betrachteten Communities sehr unterschiedlich. Tabelle 8.2 stellt die Ergebnisse gegenüber.

112 https://wiki.archlinux.org/index.php/Arch_terminology#Package_maintainer, letzter Aufruf 11.4.2017

Tab. 8.2: Pointierter Vergleich der Variable Strukturen und Verfahren

Dimension	Ubuntu	Debian	Arch
entscheidende Akteure	<i>Ubuntu Member</i> und <i>Ubuntu Developer</i> bestätigen die vom SABDFL ernannten Gremien <i>Technical Board</i> (TB) und <i>Community Council</i> (CC). Spezifische <i>Teams</i> .	Einzig <i>Debian Developer</i> (DD) sind stimmberechtigt für die Wahl des <i>Debian Project Leader</i> (DPL) und bei <i>General Resolutions</i> (GR). <i>Technical Committee</i> (TC). Spezifische <i>Delegations</i> .	<i>Arch Developer</i> führen das Projekt. <i>Trusted User</i> verwalten <i>Community Repositories</i> .
Entscheidungsfindung	TB und CC diskutieren und entscheiden aktuelle Angelegenheiten, SABDFL führt, wenn es keinen klaren Konsens gibt. Punktuell top-down.	DD koordinieren sich dezentral und autonom. Projektübergreifend koordiniert DPL. TC berät und entscheidet, wenn nötig. DD können GR einberufen und alle Entscheidungen <i>überschreiben</i> .	Die zentralen Entscheidungen treffen die <i>Arch Developer</i> in gegenseitiger Absprache. Die <i>Trusted User</i> (TU) koordinieren sich untereinander.
Entscheidungsräume	Community-Angelegenheiten: CC; technische Fragen: TB. <i>User Experience</i>	einzelne Pakete: DD und <i>Debian Maintainer</i> ; projektübergreifend: DPL, wenn nötig, TC: „technische“, GR: „politische“ Entscheidungen	Maßgeblich <i>technische</i> Entscheidungen. Richtungsweisung durch <i>Arch Developer</i> . <i>Trusted User</i> koordinieren <i>Community-Repository</i> und AUR.
Legitimation	SABDFL: Financier und charismatischer Herrscher, ernennt CC und TB. <i>Ubuntu Member</i> und <i>Ubuntu Developer</i> : soziales oder technisches Engagement.	DD: <i>technische Kompetenz</i> , Kenntnis der <i>Prinzipien, Zukunftsvisionen</i> . DPL: Wahl.	<i>Arch Developer</i> : ausgewählte bewährte und verdiente User, TU: engagierte und bewährte User
Kriterien	„Ease of Use“, Freundlichkeit und ansprechendes Design	Stabilität und Kontinuität, Sozialvertrag, Freie Software	funktionale Einfachheit, Aktualität

Jenseits dieser Unterschiede ist allen gemein, dass sie sich als Meritokratien¹¹⁴ verstehen, als Do-ocracy, in der Recht hat, wer Fakten schafft, insbesondere durch Mitarbeit am Code – reguliert durch einen „rough consensus“ und die allgegenwärtige Möglichkeit, mit einem „Fork“ ein eigenes Projekt abzuspalten, mit anderen Entscheidungen und Entwicklungen.

In jedem Fall bildet die Struktur der Community eine Legitimation, die den verdienten Mitgliedern (Meritokratie) gestalterische Macht verleiht. Legitimation liegt auch in der Möglichkeit begründet, dass sich potenziell jede*r mit ausreichendem Engagement und Fleiß die notwendige Expertise erarbeiten und Teil der Elite werden kann. Zudem sind über die transparenten Kommunikate und die vielen verschiedenen beteiligten Individuen Transparenz und Balance der Macht gegeben. Somit erfüllen die Strukturen im Wesentlichen eine vertrauensbildende Funktion – Vertrauen nicht zuletzt in die Richtigkeit der Prinzipien und Werte, die der jeweiligen Community und schließlich der gemeinsam geschaffenen Technik zugrunde liegen.

Forks ermöglichen außerdem eine diverse Landschaft von alternativen Varianten; dem User bleibt dadurch im betrachteten Fall immer auch die Wahlmöglichkeit, auf eine andere Distribution umzusteigen. Diese sind selbst verschiedene Parallelentwicklungen von GNU/Linux-Systemen und haben selbst wiederum abgeleitete Distributionen – so, wie Ubuntu von Debian abgeleitet ist und es auch von Ubuntu und Arch weitere Ableitungen („Derivate“) gibt.

Entscheidende Akteure

Ubuntu Die *Ubuntu Member* und *Ubuntu Developer* bestätigen die Mitglieder der zentralen Gremien *Community Council* und *Technical Board* im Amt, die dann ihrerseits Entscheidungen ihres Aufgabenbereiches beschließen. Sie werden vom Gründer und Sponsor Mark Shuttleworth, dem „Self Appointed Benevolent Dictator For Life“ (SABDFL), ernannt und delegieren Teams, die bestimmte Bereiche selbstständig verwalten. Unter den Teams, die für verschiedene Themen zuständig sind, befinden sich auch ein Design-Team, das Usability-Studien berücksichtigt, und ein Community-Team, das Community-Management leistet – diese Zuständigkeiten sind im Vergleich hier einzigartig und manifestieren den Fokus auf einfache Nutzung und eine

114 hier verstanden als ein unhinterfragtes Ideal einer Herrschaft derer, die sich „verdient“ gemacht haben, entgegen dem Ursprung des Wortes aus der Kritik einer dystopischen Leistungsgesellschaft von Young (1958)

besondere Pflege der Community. Neben der Bestätigungsfunktion der Mitglieder sind auch Wahlverfahren und „Polls“ vorgesehen, diese wurden aber von den Befragten nicht thematisiert.

Während es bemerkenswert ist, dass Nicht-Entwickler*innen eine besondere formale Rolle in der Organisationsstruktur erhalten, erscheint deren Mitbestimmung doch relativ gering. Vielmehr haben die Gremien *Technical Board* und *Community Council* starke Einflüsse und werden vom SABDFL eingesetzt, der aber zusätzlich eine gewichtete Stimme und einen ständigen Sitz im *Technical Board* hat. Er tritt somit formal in den Hintergrund, hat aber dennoch über die Canonical Einfluss auf die Aktivitäten der dort angestellten Entwickler*innen. Die nicht entwickelnden *Ubuntu Member* haben folglich vor allem eine legitimierende und identitätsstiftende Funktion. Auffallend ist neben dem Community-Team die Existenz eines Design-Teams, das *User Experience* als Zuständigkeitsbereich hat, und dass – wie am Beispiel *Unity* deutlich wird (vgl. S. 159) – Dinge aus fachlichen Gründen (hier: *User Experience*) strikt durchgesetzt werden können.

Debian Neben den *Debian Developers* (DD) hat der *Debian Project Leader* (DPL) eine wichtige Rolle, da er Delegationen bestimmen kann, die ihrerseits bestimmte Zuständigkeiten und Befugnisse haben. Er setzt zusammen mit den Mitgliedern des *Technical Committee* neue Mitglieder in das Gremium ein. Die Developer wählen den DPL jährlich und können über eine *General Resolution* jede Entscheidung *überschreiben*, somit sind alle Gremien und Delegierten angehalten, im Sinne der Developer zu entscheiden. Eine *General Resolution* gilt generell als zu vermeiden, da das Verfahren aufwendig ist und eine „technisch eindeutig argumentierbare“ Entscheidung präferiert wird.

Die *Debian Developer* haben somit einen starken Einfluss, können sich aber im Allgemeinen auf ihre konkreten Pakete und Projekte konzentrieren, da der DPL und seine *Delegations* projektübergreifend den Überblick behalten und koordinieren. Die DD sind in ihren Zuständigkeiten für ihre Pakete weitestgehend selbständig und müssen sich lediglich mit anderen Developern koordinieren, die von ihren Entscheidungen durch Interdependenzen betroffen sind.

Es gibt hier aber *im Wesentlichen* keine stimmberechtigten Mitglieder ohne *Contributory Expertise* im Sinne der notwendigen Expertise für das technische Mitentwickeln an der Distribution (vgl. S. 76). Vielmehr gibt es Entwickler*innen, die diese Expertise als *Debian Maintainer* zwar haben, aber noch keine Stimmberechtigung und nur begrenzte Schreibrechte im

Repository besitzen. Daraus ergibt sich eine starke Grenzziehung zwischen Entwickler*innen und Usern, insofern User ohne *Contributory Expertise* in der Regel keinen Mitgliedschaftsstatus erlangen und somit jegliche Einflussnahme auf das Projekt an diese Form von Expertise gebunden ist. Seit 2010 gibt es bemerkenswerterweise zwar „non-uploading“ *Debian Developer*, die explizit keine Entwickler*innen sind, sondern andersweitig beitragen (*Interactional Expertise*), deren Anteil liegt aber derzeit bei knapp 3% (vgl. S. 165).

Arch Bei Arch ist die Anzahl der entscheidenden Akteure überschaubar. Die *Arch Developer* sind das zentrale Entscheidungsgremium. Es ist eine relativ kleine Gruppe (ca. 40 Leute)¹¹⁵, die sich ohne spezielle Gremien koordinieren. Sie geben die Stoßrichtung vor und verwalten die zentralen Repositories. Daneben gibt es die *Trusted User*, die sich ebenfalls untereinander koordinieren, das *Community-Repository* und das *Arch User Repository* verwalten. Hier spielen Laien, also User ohne (Contributory) Expertise, auf der Entscheidungsebene keinerlei Rolle.

Entscheidungsfindung

Ubuntu Bei Ubuntu spielt der Gründer und Sponsor Mark Shuttleworth eine tragende Rolle, da er die Mitglieder der zentralen Gremien *Technical Board* und *Community Council* selbst ernannt und als ständiges Mitglied mit gewichteter Stimme im *Technical Board* Führung übernimmt, wenn es keinen klaren Konsens gibt. In den Gremien werden dann die zentralen Entscheidungen diskutiert. Die Wahlbeteiligung der *Ubuntu Members* beschränkt sich im Wesentlichen auf die Bestätigung der Mitglieder des *Community Council*; zwar sind Umfragen und Wahlentscheidungen strukturell angelegt, darüber wurde in den Interviews aber nichts berichtet.

Manche Entscheidungen werden eher auf Basis der Expertise eines delegierten Teams top-down entschieden, anstatt auf die Meinungsvielfalt der Community zu setzen, sogar wenn dies Widerstände hervorruft (vgl. S. 156).

Ungeachtet der ausdifferenzierten Struktur werden dennoch vielerorts Entscheidungen eher unbürokratisch im Sinne des „rough consensus“ in gegenseitiger Abstimmung der betreffenden Entwickler*innen koordiniert – wie in den anderen betrachteten Communities auch. Darüber hinaus werden über Feedback-Funktionen auch User ohne *Contributory Expertise* mit einbezogen. Auch im Bugtracker wird an mancher Stelle Kritik an der *User Experience* als valider Beitrag angenommen. Auf diese Weise haben einfache

115 Siehe S. 170.

User ohne *Specialist Expertise* eine vergleichsweise starke Partizipationsmöglichkeit.

Debian Debian ist im Wesentlichen sehr dezentral organisiert und die *Debian Developer* haben eine hohe Autonomie über ihre Pakete. Die *Debian Developer* beraten sich im Kreis derer, die an einer bestimmten Entscheidung beteiligt sind. Gibt es hier Schwierigkeiten in der Koordination, berät das *Technical Committee*, gibt Empfehlungen und kann auch bindende Entscheidungen fällen. Ansonsten ist die Wahl in Debian ein sehr starkes Instrument, bei dem jeder *Debian Developer* eine *General Resolution* vorschlagen kann, bei der jeder die gleiche Stimme hat. Letzteres wird aber nach Möglichkeit vermieden, weil einerseits das Credo herrscht, dass rational technischer Konsens immer besser ist als Wahlen, und andererseits, weil die Durchführung einer *General Resolution* relativ aufwendig ist.

Unter den *Contributory-Expert*innen* werden Entscheidungen also in aller Regel untereinander ausgehandelt und nur, wenn Uneinigkeit herrscht, ein Entscheidungsgremium einberufen. Nicht-Expert*innen haben dabei eine marginale Rolle, da die Kommunikation mit den Entwickler*innen – auch über die entsprechenden Feedback-Schleifen (z.B. Bugtracker) – zumindest *Interactional Expertise* verlangt, also eine Kenntnis der Regeln und Umgangsformen der Developer Community (vgl. S. 76).

Arch In Arch läuft die Entscheidungsfindung ebenfalls über einen kurzen Austausch, bei dem sich idealiter diejenigen zu Wort melden, die sich damit auskennen und die Erfahrung haben. Die Koordination der Entwickler*innen findet maßgeblich auf zwei Ebenen statt. Zum einen gibt es den relativ kleinen und geschlossenen Kreis der *Arch Developer*. Durch die starke Geschlossenheit der Gruppe (im doppelten Sinn) und durch die pragmatische minimalistische Ausrichtung der Distribution gibt es hier einen starken Konsens.

Zum anderen gibt es den Kreis der *Trusted User*, der über Bewerbungsverfahren anderen Usern offensteht. Über neue Bewerber*innen wird hierbei innerhalb der *Trusted User* abgestimmt.

Die *Arch User* können im *Arch User Repository* nicht nur eigene Pakete einstellen, sondern auch über die Popularität einzelner Software-Pakete abstimmen. Die *Trusted User* können beliebte Pakete dann ins von ihnen betreute *Community-Repository* überführen. Die Abstimmung der User hat aber keine bindende Wirkung. Alle User können also durch ihre Aktivität, das aktive Bereitstellen von Paketen im AUR und das Bewerten anderer Pakete das Angebot von Software erweitern und beeinflussen.

Alle genannten Akteure haben dabei *Contributory Expertise*, Laien sind auf der Ebene der Entscheidungen nicht vorgesehen. User mit *Interactional Expertise* können sich aber auf Ebene des Bug Reporting mit einbringen.

In allen Fällen ist die pragmatische Ad-hoc-Koordination dominant, in der sich im besten Fall die betroffenen und erfahrenen Entwickler*innen zu Wort melden und sich auf eine sinnvolle Lösung einigen. Allerdings findet diese Art der Koordination je auf verschiedenen Ebenen statt:

- Bei Ubuntu spielen das *Technical Board* und die ausdifferenzierten Teams eine wichtige Rolle in der Entscheidungsfindung und punktuell werden auch Top-down-Entscheidungen durchgesetzt.
- Bei Debian sind es vor allem die *Debian Developer*, die sich nicht nur dezentral koordinieren, sondern auch starke Mitbestimmungsrechte haben und auch den DPL wählen.
- Arch wird vor allem von einer kleinen und exklusiven Elite der *Arch Developer* koordiniert.

Entscheidungsräume

In allen betrachteten Communities sind gewisse Entscheidungen bezüglich der *Programmierung der Software* zu treffen, die unterteilt ist auf verschiedene *Repositories*, die sich nach Bedeutung für das Projekt staffeln. Die Kernkomponenten werden üblicherweise am restriktivsten gehandhabt und nur von den Core-Entwickler*innen betreut. Entschieden wird dabei auch, welche Bugs und Feature-Wünsche in welchem Zeitfenster erledigt und welche Änderungsvorschläge (Patches) berücksichtigt werden sollen.

Darüber hinaus gibt es Entscheidungen, die die *Organisation* und die *Koordination übergeordneter Ziele* betreffen, wo die dezentrale Organisation der einzelnen Paketbetreuer*innen oder Teams nicht hinreichend ist.

In jeder Community gibt es außerdem die Möglichkeit, durch jeden einzelnen *Individualbeitrag* die Software, aber auch den Diskussionverlauf, potenziell zu beeinflussen (z.B. durch einen Patch), allerdings ohne Gewissheit, ob der Vorschlag berücksichtigt wird.

Im Folgenden greife ich vor allem die Aspekte heraus, in denen sich die Communities wesentlich unterscheiden.

Ubuntu Ähnlich wie bei Arch sind die *Repositories* in Ubuntu gestaffelt in ein von der Community verwaltetes Repository („Universe“) und ein Main-Repository, das von Canonical unterstützt wird. Ähnlich wie bei Arch gibt es auch eine Art User Repository in Form von *Personal Package Archi-*

ves. Diese spielen aber eine weniger zentrale Rolle als bei Arch, bieten aber ‚bastelfreudigen‘ Usern ein Instrument, spezielle Software einzubinden.

Auf organisatorischer Ebene bemerkenswert – und hier im Vergleich einzigartig – ist vor allem die dezidierte Zuständigkeit für Community-Angelegenheiten (*Community Council* und Community-Team). Während das Community-Team aktiv betreut und vor allem die Tools der Community entwickelt und verwaltet, befasst sich der *Community Council* mit nicht-technischen Angelegenheiten und moderiert beispielsweise Konflikte. Ebenfalls nur bei Ubuntu zu beobachten ist ein dezidiertes Team für den Entscheidungsraum *Design*, insbesondere mit Verweis auf Usability-Studien (vgl. Abschnitt 8.3.1).

Debian In Debian ist auffällig, dass zunächst die Dezentralität durch die Autonomie der einzelnen *Debian Developer* bezüglich der von ihnen betreuten Pakete stark im Vordergrund steht. Eine Unterteilung der Repositories geschieht nach Lizenzen,¹¹⁶ aber alle Pakete werden von den *Debian Developern* betreut.

Auf organisatorischer Ebene gibt es eine explizite Unterscheidung zwischen technischen Angelegenheiten, die vom Technical Committee entschieden werden, und „politischen“ Angelegenheiten, die über eine Abstimmung aller Entwickler*innen entschieden werden (beispielsweise über die Änderungen des Mitgliedschaftsprozesses).

Arch In Arch wird stark unterschieden in essenzielle Pakete, die von den *Arch Developern* betreut werden, den User und Community Repositories, die unter der Kontrolle der Trusted User stehen, und schließlich den einzelnen Paketen im User Repository (AUR), die die User für wichtig halten und selbst bereitstellen und pflegen.

Über diese technische Aufteilung der Repositories hinaus gibt es im Grunde keine besonderen Entscheidungsräume. Aus der Zuständigkeit der *Arch Developer* für die Core-Komponente ergibt sich auch ihre richtungsweisende, projektübergreifende Funktion.

Die Partizipation der Laien in Ubuntu bezieht sich formell im Wesentlichen auf den Bereich Community, also einen nicht-technischen Bereich. In Debian gibt es einen Bereich politischer Entscheidungen, die Grenze zu

116 Neben dem „Main“-Repository, das Software enthält, die den *Debian Free Software Guidelines* gerecht wird, gibt es noch „Contrib“, die freie Lizenzen haben, aber auf Software zurückgreifen, die den Standards freier Lizenzen nicht genügen, und „non-free“, was schließlich Software mit proprietären Lizenzen beinhaltet.

technischen Belangen ist aber nicht immer ganz klar zu umreißen. In jedem Fall sind entscheidungsbefugt im Wesentlichen nur Mitglieder, die sich durch *Contributory Expertise* auszeichnen – ebenso in Arch, wo der Kreis der entscheidenden Akteure aber deutlich kleiner ist und ein „nicht-technischer“ Entscheidungsbereich keine besondere Rolle spielt.

Legitimation und Kriterien

Ubuntu Der SABDFL von Ubuntu ist legitimiert durch sein Engagement und die finanziellen Ressourcen, die er einbringt und ohne die es Ubuntu nie gegeben hätte. Im Hinblick auf seine Zielorientierung eines User-freundlichen Linux-Systems – und die Hegemonie von Microsoft anzufechten – hat er ein starkes Charisma, wenngleich aber seine zentrale Stellung im Projekt von Einigen auch kritisch bewertet wird. Die Bestätigung der Gremien durch die *Ubuntu Members* und *Developers* hat vor allem eine legitimierende und auch identitätsstiftende Funktion.

Neben üblichen technischen Kriterien stechen in Ubuntu die Usability-Orientierung, die sich durch ein dezidiertes Design-Team auszeichnet, und eine besondere Berücksichtigung der Community, die ein eigenes Gremium erhält und eine Stimmberechtigung erhält, hervor. Die Adressierung von Nicht-Expert*innen spiegelt sich also klar wider in dezidierten formalen Strukturen, Community-Aufgaben und einem erweiterten Mitgliedschaftsbegriff.

Debian Die *Debian Developer* sind aufwendig ausgewählt und erhalten mit der Anerkennung als *Debian Developer* eine starke Legitimation. Über das Bewerbungsverfahren werden die Einhaltung und Unterstützung der Prinzipien des Projekts gesichert, unter denen die Fokussierung auf Freie Software ein wesentliches Element darstellt. Der gesamte Release-Zyklus fokussiert auf ein hohes Maß an Stabilität und Kontinuität; neue „Stable“-Versionen erscheinen erst nach einer längeren Testing-Phase und erst nach gründlicher Überarbeitung ohne Zeitdruck: „release when it’s ready“.

Arch In Arch erfolgt die Legitimation des höchsten Gremiums durch die Erwählung seiner Mitglieder durch das Gremium selbst. Diese genießen offensichtlich hohes Ansehen und Respekt in der Community durch ihre Expertise und ihr Engagement. Zentrales Kriterium der Distribution ist „Keep It Simple, Stupid“ – das gilt ganz offensichtlich auch für die organisationalen Strukturen in der Community.

8.3 Beiträge zum gemeinsamen Wissensprodukt

Bei den Fragen zu den verschiedenen Möglichkeiten, *Beiträge zur Distribution* beizusteuern, wurde zunächst offengelassen, welche Arten von Beiträgen hier fokussiert werden. Anschließend wurde nach weniger „technischen“ Möglichkeiten gefragt, wenn dies nicht selbst von den Interviewten thematisiert wurde. Dabei wurde für die Analyse in Betracht gezogen, was von den Interviewten benannt und wie es thematisiert wurde.

Grundsätzlich gibt es im Prinzip in allen FLOSS-Projekten dieselben Beitragsmöglichkeiten: Software-Programme schreiben und verändern durch den Beitrag von Softwarecode, dokumentieren, wie die Software programmiert wird, wie sie konfiguriert oder installiert und verwendet wird, die Übersetzung von Textfragmenten im Programm und von Dokumentation in andere Sprachen und schließlich der User Support, also eine direkte Beantwortung von Fragen über Mailinglisten, Online-Foren oder Chat-Räume (Internet Relay Chat).

Bei Distributionen ist neben dem Schreiben distributionspezifischer Software (von Installationskripten bis hin zu Administrationswerkzeugen) vor allem das Paketieren und Anpassen von Upstream-Software für die jeweilige Distribution eine zentrale Aufgabe. Hierbei werden die Software-Programme so zusammengestellt, dass User sie einfach installieren können. Dabei werden auch Abhängigkeiten von bestimmten Programmmodulen und -bibliotheken und bestimmten Versionen konfiguriert, die für die korrekte Funktionsweise erfüllt werden müssen, damit das System reibungslos laufen kann. Dies ist in den Distributionen die zentrale Arbeit der Entwickler*innen.

Abbildung 8.5 zeigt eine schematische Darstellung der Beitragsarten und eine Zuordnung verschiedener Arten von Expertise und Anwendungswissen, die für die Beitragsart notwendig sind.¹¹⁷

117 Die Beitragsart Übersetzung ist auf Abbildung 8.5 aus Gründen der Übersichtlichkeit auf die Dokumentation reduziert, findet aber teilweise auch im Software-Design der User Interfaces statt.

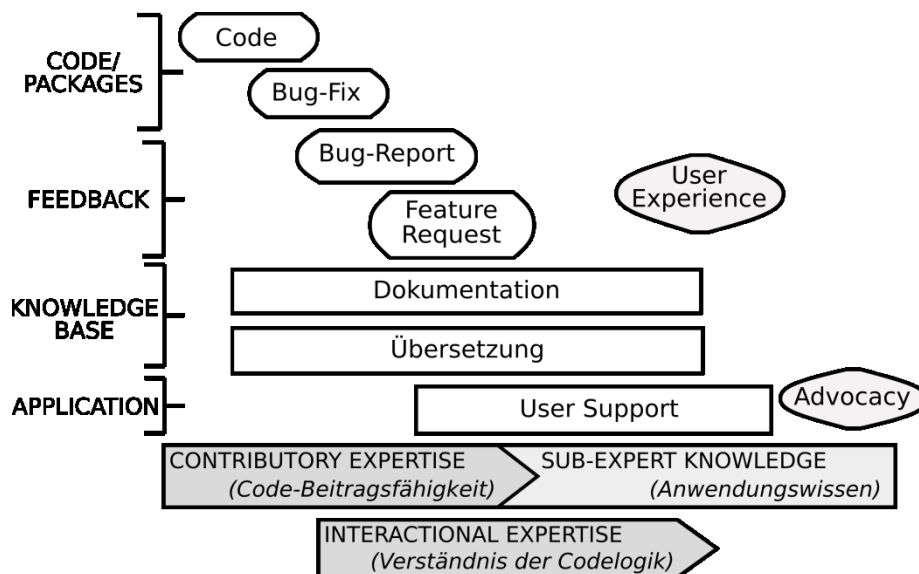


Abb. 8.5 Schematische Darstellung der Beitragsarten und grobe Zuordnung zu Formen der Expertise

Beim interaktiven Support über die genannten Kommunikationskanäle können die konkreten Anwendungsfälle, die nicht oder nur teilweise in der Dokumentation oder von der Anleitung abgedeckt werden, diskutiert werden. Dies setzt aber voraus, dass die Supporter sich in das konkrete Problem eindenken. Durch die öffentliche Kommunikation fließen die so diskutierten Fälle idealiter wiederum in das zugängliche Wissen ein. So können die interaktiv gefundenen Lösungen und die dazugehörigen Problembeschreibungen von zukünftigen Hilfesuchenden nachvollzogen werden – wenn sie die entsprechende Hilfestellung finden. Der in der Grafik der Vollständigkeit halber dargestellte User Support findet in den Interviews – und daher auch in der folgenden Beschreibung – kaum Erwähnung. Ausschließlich im Zusammenhang mit Ubuntu werden Fragen von Laien im Support als nützliches Feedback thematisiert, da Entwickler*innen auf diese Fragen teils gar nicht kommen.

In den Communities gibt es unterschiedliche Standards und Anforderungen an die Kontribution. Diese hängen zusammen mit dem vorrangigen Nutzerbild (Abschnitt 8.1) und den verschiedenen aktiven Mitglieder-Rollen innerhalb der Community (Abschnitt 8.2). Das Ergebnis der Beiträge fließt wiederum in die Community zurück und strukturiert die Wissensaufnahme der Teilnehmer*innen. Dies ist besonders offensichtlich bei Beiträgen zur

Dokumentation, zeigt sich aber auch bei der konkreten Gestaltung der Software, beispielsweise im Design der Nutzerschnittstelle (vgl. Kap. 9).

An dieser Stelle fokussiere ich die spezifischen Schwerpunktsetzungen in der Bewertung der verschiedenen *Arten von Beiträgen* und die damit verbundenen *Anforderungen an die Beitragenden* seitens der Community.

Die *Zuweisung der Aufgaben* spielt hierbei ebenfalls eine Rolle, da sich in den Rollenzuschreibungen und Arbeitsstrukturen normative Setzungen widerspiegeln. Zwar herrscht in FLOSS-Communities üblicherweise das Prinzip der Selbstzuweisung der Aufgaben, dennoch gibt es gewisse Community-spezifische Abläufe – und gerade für Neueinsteiger ist es teils nicht ganz offensichtlich, mit welcher Aufgabe man sinnvoll einsteigen kann. Dies wird in den Communities teils auch mit entsprechenden Maßnahmen adressiert.

Bei der Analyse wurde auch ein Augenmerk darauf gerichtet, ob die Art des Beitrags von den Interviewten eher als Aufgabe der *Developer* oder eher der *User* gesehen wird, was Einblicke in das Rollenverständnis innerhalb der Community ermöglicht.

Das Kapitel ist nicht entlang der Dimensionen der Variable strukturiert, sondern vorrangig nach den einzelnen Beitragsformen, da die *Anforderungen* an die Beitragenden sich nach der *Art der Beiträge* und der Rolle der beitragenden Akteure richten. Die Zuweisung der Aufgaben geschieht eher implizit. Als explizite Zuweisung für Einsteiger gibt es aber in manchen Fällen eine Kategorie von Bugs, die auf eine Art *einfach* zu lösen sein sollen. Die Dimensionen *beitragende Akteure* und die *Anforderungen an die Beitragenden* werden jeweils anschließend nochmals gesondert in einem Unterabschnitt zusammengefasst.

8.3.1 Ubuntu – „Whatever skills“: Bugs und Brötchen

In Konsequenz einer recht einsteigerfreundlichen Grundhaltung, die sich in der Variable Mitgliedschaft deutlich gezeigt hat, sind – ähnlich den Voraussetzungen, die User für die Nutzung mitbringen sollten – auch die Anforderungen an die Beitragenden relativ niederschwellig. “Usually the distros and body of open source projects make sure that, whatever is – you have skills to contribute, we will find some way to use it, you know” (4,151).

Wenngleich dies hier idealisiert als generelle Eigenschaft von Open-Source-Gemeinschaften beschrieben wird, zeigt sich in der Aussage des Ubuntu Developers ein sehr weit gefasstes Verständnis von Beiträgen, das

sich in der Anerkennung von Erfahrungsaustausch oder viralem Marketing (Advocacy) in der Ubuntu Community beobachten lässt.

Advocacy

Virales Marketing wird unter dem Begriff *Advocacy* als expliziter Beitrag gewertet und als Einstiegspunkt betrachtet, um weitere Fähigkeiten zu erwerben und tiefer in die Materie einzusteigen. Die Fürsprache und Werbung für das Projekt ist eine Beitragsform, die allein im Zusammenhang mit Ubuntu von den Interviewten explizit thematisiert wurde, und stellt eine einfache, nicht-technische Beitragsmöglichkeit dar (*Beer-Mat Knowledge*). Bei lokalen Anwendertreffen können User ihre persönlichen Erfahrungen mit Ubuntu berichten und bei „Ubuntu-Partys“ zählt sogar das Brötchen schmieren als wichtiger Beitrag (14,125). Auf diese Art findet sich im Grunde für jeden User eine Aufgabe: “User advocacy is, it’s more about sharing your experience, it’s not requiring any special skill, and then we are trying to find use for, for everyone” (4,153).

Knowledge Base

Als Beiträge werden *Dokumentation* (6,60) und auch die *Übersetzung* und Arbeiten zur *Lokalisierung* eines Projekts genannt (5,46f), dafür sind auch sprachliche Fähigkeiten fernab von technischem Detailwissen gefragt: “If your only skill is to translate in German, that’s fine” (4,152).

Feedback

User Experience Eine Besonderheit gegenüber den anderen Fällen ist, dass hier durchaus auch Mängel an der Bedienbarkeit eines Programms als Programmfehler, kurz *Bug*, berichtet werden können. Hier zählt die subjektive Erfahrung der User als wertvolle Information. Außerdem ist auch die Rückmeldung jedes Users potenziell nützlich: Die *User Experience* ist eine im Fallvergleich einzigartige Beitragsform:

Filing bugs is a very broad term in our case. because bugs are also missing functionality or something that’s not intuitive ... So if someone says that they can’t use it, because they don’t understand some linux, some – something that they have to do, device or so, or their desktop, that’s a bug that we will forward up to our design team, to evaluate, right? (5,130)

Hier wird also auch eine Kritik der Usability als valider Beitrag gewertet und anders als in anderen FLOSS-Projekten gibt es ein Team, das sich insbesondere mit *User Experience* beschäftigt. Usability-Spezialisten machen offen-

bar dedizierte Studien, wie User mit unterschiedlichen Programmfunktionen und Oberflächen zurechtkommen (5,126). “In Ubuntu staff were trained to form people that value user experience, because they won’t fit in the natural model of open source software development” (4,153).

In diesem Zusammenhang stellt der befragte Entwickler auch fest, dass es sehr wichtig ist, gerade auch negatives Feedback zur Usability zu bekommen, da es nach seinem Ermessen zu häufig passiert, dass User sich im Stillen ärgern und dann auf andere Systeme ausweichen (5,129f). Die Einstellung, Kritik aufzunehmen und User in der Community zu halten, steht hier im deutlichen Kontrast zu Arch Linux, wo eher Äußerungen auftauchen, dass User, denen Arch nicht gefällt oder nicht einfach genug ist, gerne zu einem anderen Linux wechseln sollen, zum Beispiel Ubuntu.

Bug Reporting Bei den Fehlerberichten wird versucht, die Schwelle für eine Beteiligung niedrig zu halten. Wer der englischen Sprache nicht mächtig ist, kann nach Aussage eines Entwicklers nötigenfalls auch in seiner eigenen Sprache einen Fehlerbericht einreichen:

I don’t think there is anything, other than to speaking English, for that’s the language that we all trying to speak. But even if – we try to not, you know, not to reject people that can’t file a bug. If that’s in a different language, we try to find someone that will actually be able to work with that, right? (5,52)

Bug Reports werden offensichtlich als relativ technische Beiträge doch möglichst fehlertolerant und undogmatisch hingenommen (s. auch Abschnitt 8.1.1).

Ebenfalls niederschwellig ist die technische Umsetzung der Beitragswerkzeuge, so bemerkt ein *Debian Developer*, dass der Bugtracker in Ubuntu deutlich einfacher zu benutzen ist, als dies bei Debian der Fall ist: “Again the bug reporting system is more easy to use uh, it’s easier to use, definitely, to browse the bugs” (11,127).

Ein Teil der Bug Reports erfolgt hierbei sogar durch automatisierte Meldungen des Systems, die lediglich einer Bestätigung des Users bedürfen. Aber auch „einfache Nutzer“ schreiben ausführlichere Bug Reports, was häufig komplex und relativ voraussetzungsvoll erscheint – jedoch geschieht dies tendenziell nur in bestimmten Fällen, wenn zum Beispiel ein Fehler besonders lästig ist (14,53ff).

Die Offenheit gegenüber niederschwelligen, und aus Entwicklersicht unvollständigen, Bug Reports birgt aber zusätzliche Arbeit, insofern die Bugs im Anschluss bereinigt und sortiert werden müssen. Manche Bugs enthalten beispielsweise nur Bemerkungen wie „doesn’t work, I just want it fixed“,

was es Entwickler*innen äußerst schwer macht, diese Meldungen zu berücksichtigen (6,98). Daher ist es wichtig, die neu eingestellten Bugs zu verifizieren und mit hilfreichen Informationen anzureichern, was auch „Bug Triaging“ genannt wird.

Bug Triaging Werden auch rudimentär gemeldete Softwarefehler als valide Bug Reports akzeptiert, ist das Beheben des Fehlers zunächst schwierig. Sie müssen für andere *reproduzierbar* sein, um zu verstehen, wo mögliche Ursachen liegen und um den Fehler mit einem *Debugger* zu testen und weitere Informationen zu gewinnen. Bugs treten auch häufig nur mit spezifischen Programm- oder Systemkonfigurationen auf – etwa verschiedene Hardware, Programm- oder Bibliotheksversionen, auf die die betreffende Software zugreift.

Teil des Bug Triaging ist neben dem Entfernen oder Zusammenführen von Duplikaten (mehrfach gemeldete identische Bugs), zu klären, ob das Problem tatsächlich ein Bug im Programmcode ist (6,98) oder vielleicht ein Konfigurationsfehler. Dafür sollte man in der Lage sein, in etwa zu verstehen, auf welcher Ebene das Problem liegen könnte: “In general, yeah, it’s the act to understand that if you – between like, is it a misconfiguration, is it a bug, is it something that needs to be fixed upstream, or back in our packaging” (6,69).

In der Masse der Fehlerberichte ist das Ignorieren von unvollständigen oder unklaren Fehlerbeschreibungen durchaus naheliegend und auch übliche Praxis in FLOSS-Projekten, da das Triagieren von Bugs sehr aufwendig sein kann. Folglich passt es gut zu der oben beschriebenen Akzeptanz von unvollständigen Bugs, dass ausschließlich Developer aus der Ubuntu Community Bug Triaging als Beitragsart nennen.

Praktisch gestaltet sich das Bug Triaging aber als nicht trivial – und hier könnten noch mehr Beitragende gebraucht werden.

So even if they have not upload rights, they have the rights to – they can get the rights for triaging bugs, where you can set the state from new to – and they can ask a question and mark it incomplete. Or they can mark it confirmed, if they can reproduce it too ... so, for siffling through a few general bug reports, actually having users helping, too, that would be very helpful.

For years, we had a discussion about whether that’s – maybe not as much as a good activity for real users to do, as we had originally hoped to – cause it turned out, it’s actually very hard to properly falsify a bug, though. (6,61f)

Code and Packaging

Die nächste Stufe über dem Fehlerbericht (oder zum Bug Triaging) ist der Beitrag von Code, zunächst im Sinne von kleinen Code-Schnipseln, sogenannte Patches. Diese können entweder dazu dienen, Fehler zu beheben, oder aber neue Funktionen bereitzustellen.

Einen einfachen Patch einzubringen, ist insofern niederschwellig, als jede*r Externe einen Patch vorschlagen darf, der dann einem Review unterzogen wird (6,74f). Hierzu ist also keine formale Rolle nötig. Jedoch ist die Akzeptanz solcher Patches von Externen je Projekt unterschiedlich, was nicht zuletzt auch zusammenhängt mit der Anzahl der Patches, die das jeweilige betreuende Entwicklerteam jeweils zu bewältigen hat. Die externen Beiträge müssen bewertet und angenommen oder abgelehnt werden. Dies geschieht durch ein Peer-Review-Verfahren.

Wichtig ist dabei, dass Änderungen keine bestehenden Funktionen lahmlegen. Wenn weitere Arbeit notwendig ist, wird mit dem Beitragenden Kontakt aufgenommen, möglicherweise wird ein anderer Lösungsweg vorgeschlagen. Manchmal kommt es auch dazu, dass ein Vorschlag ganz abgewiesen wird. Aus Sicht der Entwickler*innen ist das nicht so schlimm, weil jede*r ja frei ist, dies bei Bedarf in seinen persönlichen „Fork“ einzubauen (5,93). Aus Sicht der Beitragenden ist es ärgerlich, weil die Arbeit keine Anerkennung bekommt – und wenn der Patch nicht übernommen wird, muss er nachher individuell immer wieder neu angewendet werden, nachdem eine aktualisierte Version bezogen wurde.

Das erfolgreiche Einbringen von Code-Änderungen ist bei Ubuntu im Vergleich zu Debian deutlich einfacher (4,54). Das ist zunächst gut für die Entwicklung von Ubuntu, da Ubuntu aber von Debian abgeleitet ist (*Downstream*), steht dies im Konflikt mit der generellen Prämisse, Verbesserungen am Code möglichst weit „oben“ (*Upstream*) einzupflegen, weil somit alle anderen Downstream-Projekte ebenfalls von der Verbesserung des Codes profitieren. Zudem müssen Änderungen, die nicht zurück in den Upstream geflossen sind, beim Bezug aktualisierter Versionen von dort wieder neu eingepflegt werden.

The best thing would be, to submit to Debian and let it flow into Ubuntu. The second best thing is to make it enter the canonical review process, but then also submit the package to Debian. And the worst option is to only submit it in the Ubuntu repositories.

And so the trade off is to do the second solution, which is, you can still get the software you need to put into Ubuntu into a given six month release, but you

still hope that Debian will accept your package, and ... then it will flow back like the best of works. (4,55)

Packaging Um direkt Pakete in das Repository hochzuladen, werden entsprechende Zugriffsrechte, nämlich Schreibrechte benötigt, die man erst als anerkannter Ubuntu Developer bekommt (11,110f). Dabei gibt es unterschiedliche Developer-Rollen, von Core-Developern mit Zugriffsrechten auf alle Ubuntu-Repositories, über „Masters of the Universe“ mit Zugriff lediglich auf das *Community-Repository Universe* bis hin zu *Per-Package Uploaders*, die nur ein einzelnes Paket betreuen.¹¹⁸

Jedoch gibt es – ähnlich wie bei Arch Linux – einen Bereich neben den offiziellen Repositories, in dem User eigene Softwarepakete anbieten können, das sogenannte „Personal Package Archive“, kurz PPA (6,31f; 13,42; 15,147f). Hierfür gibt es Tools, die das Einstellen von Paketen erleichtern und somit für User niederschwelliger machen. Dazu muss sich ein User lediglich einen Account auf der User-Plattform *Launchpad* einrichten und den *Code of Conduct* mit einem kryptografischen Schlüssel signieren.¹¹⁹

Auch Nischenprojekte – Software, die aufgrund der kleinen Nutzerbasis nicht in das offizielle Repository aufgenommen wird – können so über das Personal Package Archive (PPA) für die User leicht nutzbar gemacht werden (13,42). Aus Nutzersicht gibt es neben der komfortablen Verfügbarkeit im Repository zwar immer den Weg, die Quellen selbst zu kompilieren, dies ist aber doch für einfache Nutzer nicht ganz trivial. Das PPA ist eine einfachere Möglichkeit, diese nicht offiziell unterstützte Software zu installieren und zu verwenden.

Jedoch sind die inoffiziellen persönlichen Archive mit Vorsicht zu genießen (15,147f), da hier nicht die sonst für das offizielle Repository übliche Qualitätskontrolle stattfindet. Somit können instabile oder unsichere Konfigurationen entstehen. Dies wiederum impliziert ein gewisses Hintergrundwissen für die Einschätzung der damit verbundenen Risiken.

Zuweisung von Aufgaben

Da sich das Bug Reporting und Triagieren nicht immer ganz voraussetzungsfrei darstellt, gibt es bei Ubuntu – um auch hier niederschwellige Ein-

118 Ausführlicher nachzulesen auf <https://wiki.ubuntu.com/UbuntuDevelopers>, letzter Aufruf 19.5.2017.

119 Siehe auch den Status des sogenannten Ununtero in Abschnitt 8.1.1, das Signaturverfahren wird auf S. 119 näher erklärt.

stiegsangebote bereitzustellen – die sogenannten *Papercut Bugs*, die auf triviale Fehler abzielen. Diese sind zwar einfach zu beheben, aber durch die explizite Adressierung wird ihre Bedeutung aufgewertet und so wenig erfahrenen Usern eine Motivation für einen Einstieg ins Bug Fixing gegeben. “I don’t think that there should be any more preconditions to gettin’ involved with the software as an hour to get used. I think that many distros are doing a good *job* of that, I mean Ubuntu does that by their papercut bugs for instance” (3,191f).

Das englischsprachige Wiki¹²⁰ beschreibt „Papercut Bugs“ als “fast to fix, but annoying bugs”, mit dem Hinweis “Our mission is to make Ubuntu shined by reducing them“.

Eine im Vergleich einzigartige Art von Zuweisung ergibt sich durch Canonical, die bezahlte Entwickler*innen anweisen kann, an bestimmten Aufgaben zu arbeiten. Jenseits der reinen Software-Programmierung beschäftigt Canonical auch Angestellte, die mit Usability-Studien beschäftigt sind (5,118) und im Community-Team mitarbeiten (4,77). In Ubuntu können somit Aufgaben priorisiert werden, die üblicherweise in Open-Source-Projekten nicht so populär sind (4,153). Außerdem sind bestimmte Aufgabenbereiche dedizierten Teams zugewiesen, die sich in regelmäßigen Online-Meetings via IRC koordinieren (s. auch S. 163).

Beitragende Akteure

Offenbar spiegelt sich die Adressierung von nicht-technischen Usern wider in niederschweligen Einstiegsangeboten für die Kontribution und führt im Ergebnis zu mehr aktiv beitragenden Usern, die nicht über *Contributory Expertise* verfügen (13,72). Dies wird nicht zuletzt auch zurückgeführt auf die Art, wie das Beitragen durch technische Tools unterstützt wird: “I think there are more less technical users involved in Ubuntu ... because again the bug reporting system is more easy to use” (11,127). Ebenfalls gestaltet sich auch das Werkzeug zum Beitragen von Übersetzungen als relativ ansprechend (11,81) – eine Beitragsart, die an sich genommen auf rein verbalsprachlicher Ebene liegt und damit wenig technisch ist. Die Integration solcher nicht-technischer Expertise kann aber dennoch an technischer Umsetzung scheitern, wenn die Beitragenden nicht mit den angebotenen Werkzeugen zurechtkommen. Ein Faktor für die niederschwellige Gestaltung der technischen

120 <https://wiki.ubuntu.com/OneHundredPapercuts/Mission>

Tools mag auch die dedizierte Betreuung ihrer Gestaltung durch das Community-Team sein (15,65).

Generell zeigt sich hier also ein weit gefasster Rahmen von Beiträgen, bei denen auch nicht oder weniger technische Beiträge eine besondere Wertschätzung erfahren.

Anforderungen an die Beitragenden

Die Tools für den Bug Report sind relativ einfach zu benutzen und es gibt explizite Beitragsmöglichkeiten und Einstiegsangebote, die verhältnismäßig niederschwellig sind. Bemerkenswert ist die Akzeptanz auch von unvollständigen Bug Reports. Dadurch wird Usern ohne *Contributory Expertise* eine Feedback-Möglichkeit eingeräumt und durch die Kategorie der *User Experience* eine explizite Laienperspektive ermöglicht.

Daraus ergibt sich offenbar ein interessanter Effekt. Weniger technisch geschulte User dürfen Bug Reports verfassen. Dadurch ergeben sich mehr Fehlerberichte, die schwerer zu bearbeiten sind, was zunächst für die Developer ein doppeltes Problem darstellt. Allerdings entsteht eine neue Beitragsform, die in ihrer *Anforderung an die Beitragenden* zwischen dem Bug Fix (Coding) und dem Bug Report liegt: das Bug Triaging. Hier müssen User nicht in den Softwarecode eintauchen und die Fehlerursachen beheben, sondern lediglich durch Nachfragen beim Reporter und durch Testen der Software herausfinden, wann der Fehler auftritt, und den Bug Report mit Informationen anreichern. Dadurch ergeben sich *mehr Möglichkeiten* der aktiven Kontribution für User *mit weniger technischer Expertise*. Das Problem des erhöhten Aufkommens von Bug Reports kann also theoretisch durch eine größere Anzahl von Beitragenden gelöst werden, die sich durch vereinfachte Beitragsanforderungen ergibt. Praktisch fehlt es allerdings an Usern, die sich beim Bug Triaging einbringen (6,61f; s. auch S. 189).

8.3.2 Debian – Developer unter sich

In den Interviews spielen in Bezug auf Debian weniger technische Beiträge, – wie das Übersetzen von Dokumentationen und den in Software-Programmen verwendeten Texten – eine nachrangige Rolle. Eine größere Aufmerksamkeit wird hier dem Bug Reporting, der Fehlerbehebung und dem Packaging gewidmet.

Knowledge Base

Übersetzung Als Beitragsmöglichkeiten nennt ein Entwickler zuallererst Übersetzungen, gerade auch als Beitrag nicht-technischer Nutzer*innen (12,7). Allerdings stellt ein anderer fest, dass dies häufig in erster Linie durch das Team der betreuenden Entwickler*innen geschieht. Nur bei sehr prominenten Projekten gelingt es demnach, größere Teams dezidiert für die systematische Übersetzung der Textinhalte von Software-Programmen anzusetzen (16,268f).

Dokumentation Dokumentation ist eine stetige Aufgabe, zu der explizit alle aufgefordert sind. So stellt im Laufe einer Mailinglisten-Diskussion über die Legitimität des Akronyms RTFM¹²¹ der damalige Debian Project Leader fest: “When the code is public, RTFM is the proper answer. One might add: document it properly afterwards”¹²².

Damit wird festgestellt, dass die Öffentlichkeit und Zugreifbarkeit des Softwarecodes idealiter jedem oder jeder die Möglichkeit bietet, sich bei Fragestellungen, die nicht dokumentiert sind, in den Quellcode einzulesen. Dort ist schließlich alles festgeschrieben und jede Funktionalität, Konfigurierbarkeit und Bedienungsmöglichkeit ist grundsätzlich ableitbar.

Andersherum betrachtet, stellt die Aussage aber auch noch einmal klar, dass Dokumentation nicht das erste Ziel und die größte Sorge der Entwickler*innen ist, eben weil aus deren Perspektive alles im Code zu finden ist. Folglich ist es nicht weiter verwunderlich, dass sich die Verfügbarkeit von Dokumentation, insbesondere in Bezug auf Aktualität und Korrektheit, teilweise recht dürftig darstellt. “But there when you look on *how* to compile a debian package, you will find a ton of documentation and all of them are going to be deprecated. – It’s, it’s hard to find some good documentation, and, you have *one*, you *have* to find it” (2,103). Dies wiederum erschwert den Einstieg für diejenigen, die sich erst einarbeiten müssen (2,97ff) und stärkt wiederum die Bedeutung von Mentor*innen, die einem Hinweise geben, wo man die Dokumentation findet, oder einem gegebenenfalls auch direkt erklären, wie etwas funktioniert (16,86ff).¹²³

121 „Read the Fucking Manual“, siehe oben.

122 <https://lists.debian.org/debian-vote/2005/03/msg00610.html>, letzter Aufruf 19.5.2017. Diese Passage wird auch zitiert in Coleman (2013: 111).

123 Bemerkenswert ist hierbei, dass auf einem Vortrag auf den Chemnitzer Linux-Tagen 2016 zu Debian als hilfreiche Links explizit auf die Wikis von Arch und Ubuntu

Feedback

Bug Reporting als Feedback Als eine wichtige Form von Feedback wird das Bug Reporting betrachtet, da es zentral ist, um Probleme zu identifizieren und das Produkt zu verbessern.

“I think the more interesting thing is bug reporting, I think that is the most important – we need feedback from, what’s broken in the distribution, it’s for a regular user I’d say, bug reporting is really important I mean, you can easily report bugs to the distribution.” (11,78f)

Einfach ist hierbei relativ, da in diesem Kontext anschließend kontrastiert wird, dass dieser Schritt bei Ubuntu deutlich einfacher organisiert ist. Jenseits der technischen Umsetzung des Beitrags gibt es bei Debian auch strikte Vorgaben, wie der Bug Report auszusehen hat, im Gegensatz zur oben berichteten Fehlertoleranz von Ubuntu.

At the start its very difficult, because you have to learn a lot of processes, tools. Every project has a website where you can create bugs – and here you have to use some common line, some report tool, the report bug tool. But that’s not obvious, but after a while you get used to it and it feels like a superior solution. So yes, there is quite a high entry cost. (16,83)

Interessant ist hierbei auch, dass der Interviewte die komplizierte Vorgehensweise retrospektiv als eine überlegene Methode betrachtet.

Darüber hinaus kann auch auf niederschwelligere Weise Feedback gegeben werden, wie ein Entwickler auf die Frage nach Beitragsmöglichkeiten reiner User bemerkt. Diese stellen aber nicht das übliche Prozedere dar: “Sometimes they find bugs, yeah. Sometimes even if they can’t register the bug themselves, they give you a mail or complain on an irc, and then you can find the bug yourself” (16,186f).

Code and Packaging

Besteht also seitens eines Users ein starkes Interesse daran, bei der Lösung von Problemen mitzuwirken, oder der Impuls, einem verwaisten Paket etwas mehr „Liebe“ (16,71) zukommen zu lassen, bietet es sich aus strukturellen Gründen an, eine formelle Mitgliedschaft anzustoßen. Sind Beitragende sehr aktiv, werden sie dem Maintainer des entsprechenden Pakets auffallen. Da

(Ubuntuusers.de) verwiesen wird, das Debian-Wiki aber keine Erwähnung findet, wenn auch die User-Mailingliste und die Forum-Website von Debian genannt werden. Siehe <https://chemnitzer.linux-tage.de/2016/de/programm/beitrag/> 191 (ab Minute 53), bzw. Folie 47 <https://people.debian.org/~andi/Chemnitz2016.pdf>, letzter Aufruf 28.3.2017.

der Maintainer den Code selbst einpflegen muss, besteht ein Anreiz, aktive Beitragende in eine formale Rolle zu bringen. Für die Betreuung eines einzelnen Pakets reicht schon die erste Stufe des *Debian Maintainer* aus. Aber auch diese Stufe ist schon mit einem relativ aufwendigen Qualifizierungsverfahren verbunden (13,41), da hier auch Schreibrechte im Repository erteilt werden (s. auch Abschnitt 8.1.2). Ein naheliegender Schritt für *Debian Maintainer* ist, dann die zweite Stufen zum *Debian Developer* anzustreben, um auch ein Stimmrecht in der Community zu erhalten, was jedoch mit einem weiteren Bewerbungsverfahren verbunden ist.

Bemerkenswert ist in dieser Hinsicht, dass neben der auch andernorts üblichen Praxis, eine*n Fürsprecher*in („Sponsor“) in der Community zu haben, der oder die einen unterstützt und das bisherige Engagement bestätigt, auch eine Art Prüfung über Themen wie „Freie Software“ und technisches Grundwissen stattfindet:

And I thought myself, let's package that in Debian and Ubuntu, what was related. And then I started doing the thing to become a Debian Developer to – you have to pass – not an exam but you have some questions about Free Software, you have to have technical skills that need to be tested and stuff like that. And then I became a Debian Developer. (11,16f)

Zuweisung von Aufgaben

Ein Debian-Entwickler stellt fest, dass die Zuweisung von Aufgaben grundsätzlich nicht zur FLOSS-Philosophie passt: “The rule of the software contract is you can't say someone you must do that, everyone does what he wants, so when you have someone that comes ‘what should I do?’ – We are not prepared. We have not a hierarchy where is a boss that says you must do that” (16,171f).

Vor diesem Grund stößt die Frage von Usern nach einer Möglichkeit des Beitragens auf wenig Verständnis. Vielmehr geht dem aktiven Beitragen in ihrer Vorstellung ein konkretes Problem voraus – und daraus ergibt sich der Wille, es zu lösen. “You have to have the *will* to do it. A problem to solve it – if you see a little annoyance and think ‘I want to fix it’” (16,152).

Ist dieser Wille vorhanden, gibt es verschiedene Stufen des aktiven Beitragens von Code – vom Code-Schnipsel in Form eines Patches bis hin zum regelmäßigen Einpflegen von Änderungen, was wiederum mit einer entsprechenden formalen Rolle verbunden ist oder durch einen *Debian Developer* vermittelt geschehen muss.

Folglich kann es passieren, dass die Frage nach einer Beitragsmöglichkeit mit dem Verweis auf ein verwaistes Paket beantwortet wird – also ein Paket, dessen Maintainer sich nicht mehr aktiv darum kümmert. “For Debian it’s like, get on the mailing list and have someone hand you a deprecated *package* ((Debian-Kollege lacht)) *that sounds so exciting* ((gemeinsames Gelächter))” (2,92). Der Einstieg in die Kontribution erscheint in der Aussage bei Debian daher nicht unbedingt einfach oder attraktiv, was aber an der dahinter liegenden Hacker-Logik liegt, die nicht Partizipation als abstraktes Ziel hat, sondern auf individueller Aneignung als konkrete intrinsische Motivation basiert.

Dennoch gibt es Bestrebungen, auch Einstiegsangebote für User anzubieten. Eine Variante davon sind Bugs, die explizit für Einsteiger*innen gedacht sind, sogenannte „Newcomer“-Bugs (2,94). Leicht ist dabei aber auch relativ, beispielsweise versteht ein Entwickler darunter das Aufsplitten eines Software-Pakets in zwei Teile – für ihn eine einfache Aufgabe. Sein Entwicklerkollege, der nicht in Debian aktiv ist, empfindet dies hingegen schon als eine komplizierte Aufgabe (2,95ff) – insbesondere, wenn man die entsprechende Dokumentation nicht findet (s.o.). Die Institution dieser Kategorisierung zeugt aber von einem Interesse innerhalb der Community daran, Neueinsteiger*innen zu gewinnen, deren Erfolg aber im Interview als mäßig betrachtet wird, da die Newcomer-Bugs offenbar keinen großen Anklang finden (2,94).

Ein anderes Einstiegsangebot orientiert sich anwenderorientiert an den Softwarepaketen, die auf dem entsprechenden System installiert sind. Das Programm „how-can-i-help“ analysiert die installierten Pakete und zeigt eine Liste von aktuellem Unterstützungsbedarf an.

Jenseits der dezentralen Verwaltung der Pakete durch ihre Maintainer gibt es eine Reihe von Aufgaben, die zentral vom DPL auf jeweilige *Delegations* verteilt werden. Diese werden von ihm ernannt.¹²⁴ Delegierte können aber auch nur *Debian Developers* sein.

Beitragende Akteure

Im Gegensatz zu Ubuntu – wo, wie ausgeführt, durchaus auch nicht-technische User involviert sind (s.o.) – formuliert ein User, dass bei Debian auch nicht-technische Beiträge wie beispielsweise Übersetzungen und Anleitungen (Dokumentation) häufig von technischen Usern beigesteuert werden.

¹²⁴ Aufgrund der Freiwilligkeit muss dies in Rücksprache mit den entsprechenden Mitgliedern geschehen und stellt daher eine freiwillige Arbeitsteilung dar.

With Debian I don't know if there is much non-technical contribution, I mean there – of course there is a wiki, there is documentation, there is art work, but along that's being done *by* the technical users, whereas Ubuntu actually seems to have people who are writing that sort of content who aren't necessarily as technical. (13,72)

Grundsätzlich kann auch hier jeder User Beiträge für das Projekt generieren und auch hier gibt es Anleitungen für das Wiki, Übersetzungen und Mailinglisten und Foren, in denen Support für andere User gegeben wird. Insbesondere sind Wikis, Foren und Mailinglisten offene Bereiche, für die man sich zwar registrieren muss und gegebenenfalls ein Online-Konto benötigt, aber der Zugang ist hier nicht in vergleichbarem Maße beschränkt wie bei der Code Basis, dem Repository. Code muss aber eindeutig und fehlerfrei sein, um zu funktionieren – im Gegensatz dazu kann das dokumentierte Wissen, wie die Programme beispielsweise verwendet werden, durchaus diskutiert werden und ist zumindest für dessen Funktionsweise unkritisch.

Auch gibt es freilich Debian User, die in Linux User Groups ihre Erfahrungen und ihr Wissen teilen, jedoch findet dies nicht als „Advocacy“ explizite Erwähnung in den Interviews – außer in negativer Abgrenzung durch einen Ubuntu-Entwickler: “Debian doesn't do this advocacy part: reaching up, mentoring people, whatever we said” (4,173).

Anforderungen an die Beitragenden

Im recht komplizierten Bewerbungsverfahren zeigen sich also schon hohe Anforderungen an die technischen Fähigkeiten der Entwickler*innen, aber auch an deren Kenntnisse über die Debian-spezifischen Prozesse und Strukturen sowie die als grundlegend erachteten Richtlinien Freier Software. Nicht zuletzt ist, wie oben ausgeführt, die Formulierung von Visionen – also Zielen, die über eine unmittelbare Problemlösung hinausgehen – eine Voraussetzung für ein erfolgreiches Mitgliedsverfahren.

Die Kommunikation findet maßgeblich über Mailinglisten statt (2,84; 2,90f), was manchen als unattraktiv oder gewöhnungsbedürftig erscheint und was mit einer strengen Diskussionskultur verbunden ist. “Debian is tricky, debian is a tricky project, cause everything is happening on the mailinglist, which is quite hard at the beginning, and some people can be rude, and Debian – it is hard to start contributing to Debian” (2,84).

Für die korrekte Formulierung eines Bugs beispielsweise sind einige Regeln zu beachten, was sich in der Außenwahrnehmung als bürokratisch darstellt. Debian Policies haben demnach einen hohen Stellenwert – was von

Beitragswilligen teils als Widerspenstigkeit aufgefasst wird: “Usually when you propose your package to Debian they will have comments about it – like, ‘you’re packaging there as you shouldn’t be doing that, because according to our (in strengem Tonfall) *rules*, which you have *ignored*, you should do this *this way ...*” (4,62).

8.3.3 Arch – Userdeveloper und Developeruser

Aufgrund der beschriebenen Fokussierung auf „kompetente und engagierte“ Nutzer wenig überraschend, spielt die Niederschwelligkeit von Beitragsmöglichkeiten eine geringere Rolle. Vielmehr wird hier davon ausgegangen, dass jede*r sich das notwendige Wissen selbstständig erarbeiten kann. Außerdem gab es Äußerungen, die sich – ganz im Gegensatz zu Ubuntu oder Debian – eher wenig glücklich über eine zunehmende User-Basis und Verbreitung des Systems zeigten. Vielmehr wurde die Ausbreitung mit einem gewissen Qualitätsverlust in Verbindung gebracht, der insbesondere in der Dokumentation im Arch-Wiki festgestellt wurde.

Auch in Arch gibt es Beitragsmöglichkeiten, die grundsätzlich ein weniger tiefes Verständnis des Systems benötigen. Prominent wurden aber insbesondere Bug Reports als Beitragsform, die ein grundlegendes technisches Verständnis verlangt, genannt.

Knowledge Base

Dokumentation Das Arch-Linux-Wiki wird von einem User als eine ausgezeichnete Wissensquelle beschrieben, weil durch die Notwendigkeit, das System komplett selbst zu konfigurieren, User dieses Wissen auch in die Dokumentation einfließen lassen. Darüber hinaus stellt er fest, dass die User, die das Wiki pflegen, eben auch den entsprechenden technischen Sachverstand haben.

I use it and I really love their wiki, because you have a lot of people who are – you know, the nature of Arch is you are configuring your own system. Because you’re configuring your own system, you get people who are *really* technical and in-depth and write, like this *very* very thorough wiki. (13,109)

Andererseits beklagen Entwickler*innen die wechselhafte Qualität des Wikis, was sie auf die wachsende Beliebtheit von Arch zurückführen und die damit verbundene erhöhte Anzahl an Usern, die nicht so technisch versiert sind (8,247).

Folglich findet sich auch in der Beschreibungsseite des Arch-Wiki der Hinweis darauf, dass die wichtigste Aufgabe ist, die favorisierten Wiki-Seiten zu beobachten, damit die eingebrachte Substanz und Qualität nicht durch kontraproduktive Veränderungen verschlechtert wird: “[T]here is one [task] that you should consider far more important than the others: add your favorite articles to your watchlist and protect them against counter-productive edits.”¹²⁵

Wechselhafte Qualität wird auch in Bezug auf den Arch Linux Internet Relay Chat (IRC) attestiert, allerdings mit Beschränkung auf den deutschen Kanal. Dort wird beklagt, dass ab und an ein paar Spaßvögel unterwegs seien, die, unter dem Radar der Community, gerne auch mal Usern falsche Hinweise geben, wenn diese offensichtlich nicht so viel Ahnung haben, weil sie es *lustig* finden (8,208ff).

Übersetzung Ein Interviewter merkt an, dass es in Italien eine sehr aktive Community bezüglich der Übersetzung gibt (1,89), was er als sehr positiv bewertet. Hier sieht er eine wichtige Möglichkeit, um Sprachbarrieren abzubauen (1,93). Bemerkenswert ist an dieser Stelle die Offenheit, die Sprachbarrieren zu überwinden, während an anderer Stelle Hilfe für die Überwindung der technischen Barrieren eine geringere Rolle spielt.

Feedback

Bug Reporting Im Kontrast zum Fall Ubuntu stehen die Anforderungen an das Bug Reporting. Auf die Frage nach Anforderungen für das Beitragen antwortet ein Interviewter gar mir dem Bedarf von „ein wenig“ Computer-Science-Wissen als Bedingung, um ableiten zu können, wo die eigentliche Fehlerursache eines Programms zu verorten ist, wie beispielsweise bei einem Programm wie dem Internet-Browser Firefox.

[You need] a bit of computer science knowledge just to know these bugs are caused – I mean you could get five obstructions were to start: Is my windows manager, is it Firefox itself, is it a plugin inside Firefox, is it my graphic card ... too many stuff and you have to know charts according to the bug ... [when you just] say “Firefox crashes”, no one will ever fix that. (1,97)

Auch in anderen Nennungen des Bug Reporting zeigt sich, dass dafür eine Auseinandersetzung mit dem auftretenden Problem erwartet wird (8,45) oder Bugs andernfalls aus Mangel an Informationen nicht weiter bearbeitet werden (7,80).

¹²⁵ https://wiki.archlinux.org/index.php/ArchWiki:Contributing#The_most_important_task, letzter Aufruf 31.3.2017

Code and Packaging

Bug Fixing Zentral ist bei Arch Linux die Orientierung am Upstream. Das heißt, dass die bereitgestellte Software mit möglichst wenigen Änderungen angeboten und mit nur minimalen Änderungen aus dem originären Projekt übernommen wird (8,84). Diese Design-Entscheidung führt dazu, dass der Paketierungs-Aufwand auf der Downstream-Ebene der Distribution minimal ist und die Fehler, die gefunden werden, nach Möglichkeit wieder an den Upstream weitergegeben oder gegebenenfalls auch dort behoben werden. „Die meisten Bugs sind halt tatsächlich Bugs in den Upstream-Projekten selbst – es ist eher selten, dass es ein Bug ist, den wir selber da eingebaut haben ...“ (8,87).

Um in Bugs auch auf Upstream-Ebene tiefer einzusteigen und dort Fehler zu beheben, benötigt es schließlich nicht nur Programmierfähigkeiten, sondern auch die Kenntnis der jeweils verwendeten Sprache und bestenfalls auch Wissen über die (organisatorischen und technischen) Zusammenhänge des Projekts (8,63). Aufgrund des Aufwands, sich in ein Projekt einzuarbeiten, und da sich der Code eines Projektes schnell ändern kann, ist es unrealistisch, sich in einer Vielzahl unterschiedlicher Projekte aktiv am Code zu beteiligen (8,104).

Mit der Upstream-Orientierung der Community geht also einher, dass das Beheben von Bugs auf die Upstream-Ebene fokussiert ist. Damit sind ein tieferes Wissen und spezifische Anforderungen der originären Projekte verbunden.

Packaging: The Arch User Repository Eine prominente Rolle, um für das Projekt beizutragen, stellt das *Arch User Repository* (AUR) dar. Das AUR ist eine Plattform, auf dem jeder User eigene Pakete bereitstellen kann. Hier können User aus der Fülle an existierenden Softwareprojekten die Pakete erstellen, die nicht in den offiziellen Repositorien verfügbar sind.

Das Team der Core-Entwickler*innen kümmert sich maßgeblich um die Kern-Komponenten des Systems, während die Trusted User als User, die sozusagen von den Core-Entwickler*innen das Vertrauen ausgesprochen bekommen haben, die weniger zentralen Software-Programme in einem *Community*-Repository pflegen. Da Bereitstellung und Pflege aber auch mit einem gewissen Aufwand verbunden sind, können die Entwickler und Trusted User nicht alle verfügbaren Programme als Pakete bereitstellen.

Zwar kann jeder User immer auch selbst die Softwarequellen eines Projektes herunterladen und selbst auf seinem Computer in ein ausführbares

Programm kompilieren, jedoch stellt die Verfügbarkeit als Paket einen deutlichen Komfort dar, da hier die Kompilierung und Integration in einer Art Baukasten zur Verfügung gestellt werden. Somit kann die Installation mithilfe von Skripten automatisiert werden und Aktualisierungen werden vom oder von der Paketbetreuer*in bereitgestellt. Die Verfügbarkeit eines Programms im AUR stellt einen deutlichen Komfort für andere User dar, die sich über das AUR so gegenseitig arbeitsteilig Nischenprojekte zur Verfügung stellen. Wenn ein derartig bereitgestelltes Paket sich einer gewissen Beliebtheit erfreut – User können im AUR für dort bereitgestellte Pakete votieren –, steht es den Trusted Usern offen, dieses Programm in das offizielle *Community-Repository* zu übernehmen (8,236).

Außerdem können sich User über die Bereitstellung von Paketen profilieren und dabei Engagement und technische Expertise im Umgang mit Paketen zeigen, was für eine formale Rolle als Trusted User ein ausschlaggebendes Kriterium ist.

Bemerkenswert ist, dass mit der Institution des AUR die Entwicklertätigkeit im Sinne einer Distribution somit explizit dem User zugerechnet wird, der dadurch auf eine Stufe mit den Entwickler*innen gestellt wird.

Zuweisung der Aufgaben

Explizite Einsteiger-Aufgaben gibt es nicht, wenngleich ein befragter Entwickler bemerkt, dass Einsteiger*innen sich sinnvollerweise weniger kritischen „easy, low priority bugs“ zuwenden könnten. Ein anderer Entwickler benennt explizit das Bug-Reporting-System als Aufgabenzuweisung für ihn als Entwickler, da er ohne Bug auch keine Veranlassung hat, Veränderungen am Code vorzunehmen. Somit kommt dem Bug Reporting eine gewisse Aufgabenzuweisung der User an die Entwickler*innen zu. „It’s important *because* – mainly because developers are lazy ... myself included I, usually I report few bugs, because I just live with bugs” (1,196).

Beitragende Akteure

An dieser Stelle sei nochmals angemerkt, dass die Rollen *Trusted User* und *Arch User* beide dezidierte Entwickleraufgaben tätigen, nämlich die Bereitstellung und Pflege von Softwarepaketen. User werden also schon durch diese Namensgebung als potenzielle Entwickler*innen betrachtet. Nichtsdestotrotz gibt es die Staffelung von Core Repositories, die von den *Arch Developern* gepflegt werden, und Community und User Repository, die von den Trusted Usern beziehungsweise den Usern selbst gepflegt werden.

Im Wiki beispielsweise gibt es keine derart restriktive Vergabe von Schreibrechten, daher kann hier jede*r mitmachen. Durch die oben beschriebene Selektion von versierten Usern – Usern mit technischer Expertise oder zumindest User auf dem Weg dorthin – sind diese hier, zusammen mit den Entwickler*innen, die wesentlichen beitragenden Akteure.

Anforderungen an die Beitragenden

Die völlige Offenheit des *Arch User Repository* (AUR) bringt aber mit sich, dass es darin keine institutionalisierte Qualitätskontrolle gibt. Daher sind einerseits die Beitragenden zur Sorgfalt aufgerufen, außerdem sollten aber die User, die auf diese Paketquellen zurückgreifen, auch selbst die Fähigkeiten mitbringen, um die dort zur Verfügung gestellten Softwarepakete zu prüfen, um nicht die Stabilität oder gar die Sicherheit ihres Systems zu gefährden.

... Im AUR kann jeder, da kann jeder irgendwas hochladen, das ist der Punkt, den man immer wieder wiederholt, bis dann neuen Usern – wenn du dir hier was runterlädst, guck dir das an, das ist wie ein Wiki, da kann jeder auch Mist reinschreiben, oder halt auch Böswilliges. (8,324)

In Bezug auf das AUR sind also schon die *Anforderungen an die User* relativ hoch, da potenziell jede*r eingeladen ist, hier beizutragen – auch, weil ebenfalls jede*r dazu angehalten ist, die dort angebotene Software kritisch zu überprüfen. Die Anforderungen an die Bug Reports sind ebenfalls relativ hoch; beziehungsweise werden Bug Reports, die keine ausreichenden Informationen beinhalten, von den Entwickler*innen nicht beachtet. Das Bug Fixing wird sehr stark auf die Upstream-Projekte fokussiert, was nach Angaben der Entwickler*innen eine nicht zu vernachlässigende Einarbeitung in das jeweilige Projekt bedingt. Die hohen Anforderungen ans Wiki zeigen sich in kritischen Äußerungen von Entwickler*innen bezüglich der abfallenden Qualität durch das Wachstum der Community, einem Aufruf im Wiki zur kritischen Überwachung von Seiten sowie in der Referenz auf das Arch-Wiki auch in anderen Communities als wertvolle Wissensquelle.

8.3.4 Vergleich der Variable „Beiträge“

Die meisten Arten von Beiträgen sind in allen Projekten vorhanden, allerdings verknüpft mit unterschiedlichen Anforderungen und unterschiedlich bewertet. Während für Beiträge zu Feedback und Dokumentation keine formalen Bedingungen bestehen, sind für Beiträge zum Softwarecode stärker

formalisierte Rollen und Zugriffsrechte notwendig. Die Adressierung und Selektion der Mitglieder durch implizite Nutzerbilder und explizite formelle Bewerbungsverfahren haben schließlich eine besondere Relevanz in Bezug auf die Beiträge, da die Anforderungen, die an die Mitglieder implizit und explizit gestellt werden, auch die Qualität der Beiträge beeinflussen.

Es folgt eine vergleichende Betrachtung der Fälle mit den auffälligen Besonderheiten. In Tabelle 8.3 kontrastiere ich insbesondere die Unterschiede zwischen den Fällen in Bezug auf die Variable Beiträge.

Tab. 8.3: *Pointierter Vergleich der Variable Beiträge*

Dimension	Ubuntu	Debian	Arch
Art der Beiträge (Besonderheiten)	Advocacy, User Experience, Bug Triaging	besonders Bug Reporting und Packaging, auch Dokumentation	Packaging, Bug Reporting, Upstream, renommiertes Wiki
Zuweisung von Aufgaben	Teams, Canonical. Einsteiger: „Paper-cut Bugs“; Bugtracker (<i>Launchpad</i>)	<i>Delegations</i> ; dezentrale Autonomie der <i>Debian Developer</i> . Einsteiger: <i>Newcomer Bugs</i>	Bugtracker, Eigeninitiative
beitragende Akteure	„more less technical users involved“	maßgeblich <i>Debian Developer</i> und <i>Debian Maintainer</i> , auch User	Developer, <i>Trusted User</i> , User
Anforderungen an Beitragende	<i>Beer-Mat Knowledge</i>	Knowledge Base: <i>Popular Understanding</i> , Feedback: <i>Interactional Expertise</i> , Debian Policies	Knowledge Base: <i>Primary Source Knowledge</i> ; Feedback: <i>Interactional Expertise</i>

Arten der Beiträge

Ubuntu *Advocacy* findet anders als bei *Ubuntu* in den Fällen von Arch und Debian keine Erwähnung, obwohl diese auch auf Linux-Messen und Entwicklerkonferenzen mit Ständen vertreten sind. Dies wird aber explizit als valider Beitrag gesehen, was auch die (völlig untechnische) Unterstützung durch Verpflegung mit einschließt.

Als Beitragsmöglichkeit ohne Expertise im Sinne eines technischen Verständnisses der zugrunde liegenden Funktionalität gilt *User Experience* als valides Feedback auch in Form von Bug Reports. Dies zeigt nicht nur eine Fokussierung auf Usability an sich, sondern darüber hinaus die damit verbundene Wertschätzung der subjektiven Perspektive von Laien auf die Benutzbarkeit der Software.

Generell wird eine Fehlertoleranz bei Bug Reports berichtet, aber auch, dass die Nacharbeit der Bug Reports, das sogenannte Bug Triaging, von zentraler Bedeutung ist. Die Perspektive der Laien muss also durch Community-Mitglieder, die *Interactional Expertise* besitzen, so aufbereitet werden, dass die Entwickler*innen damit etwas anfangen können. Hier ergibt sich durch die geringere Anforderung an den einzelnen Beitrag eine weitere Beitragsform und eröffnet zumindest theoretisch ein erhöhtes Kontributionspotenzial.

Debian In *Debian* finden sich vorrangig die klassischen Beitragsarten eines FLOSS-Projekts von Coding/Packaging über Feedback (Bug Reporting) bis zur Weiterentwicklung der Knowledge Base.

Bemerkenswert ist, dass einerseits eine Dokumentation nicht immer leicht zu finden ist, während offenbar die Antwort auf alle Fragen letzten Endes auch immer im Code steckt – “where the code is public, RTFM is the proper answer”. Offensichtlich ist für viele eine stärkere Dokumentation nicht notwendig, da sie ausreichend Expertise haben oder es nicht für nötig halten, erarbeitetes Wissen für andere festzuhalten. Der Brückenschlag zu Laien-Usern bleibt an dieser Stelle aus.

Eine gewisse Gegentendenz zeigt sich in der zunehmenden Wertschätzung von Beiträgen, die keine *Contributory Expertise* im Sinne von Code-Beitrag implizieren. Aktive Mitarbeit auf der Ebene Feedback und Knowledge Base (Dokumentation) gelten mittlerweile *formal* auch als qualifizierend für den Status als *Debian Developer*.

Arch Packaging ist hier die zentrale Art des Beitragens, auch für User. Eine besondere Rolle spielen die Softwarepakete der User im AUR. Sie liefern eine bunte Vielfalt an Software, die die Entwickler gar nicht alleine anbieten könnten, und erfüllt eine Funktion der Nachwuchsförderung.

Die starke Orientierung am Upstream führt dazu, dass Bug Reporting und Bug Fixing sich häufig außerhalb der Distribution abspielen. Dies fällt zusammen mit Usern, die tendenziell *Contributory Expertise* haben und dadurch auch im Upstream Bugs reporten oder dort gar Fehler beheben können.

Bei Arch hat das Wiki einen sehr guten Ruf, da die Dokumentation dort offenbar von Usern geschrieben ist, die eine gewisse Expertise besitzen.

Zuweisung von Aufgaben

Üblicherweise herrscht in FLOSS-Projekten eine Selbstzuweisung der Arbeit, die über Bugtracker koordiniert und die von den eigenen Interessen der Mitglieder geleitet wird. Die Community birgt aber ein starkes Identifikationspotenzial, das dazu führt, dass sich die Mitglieder die Ziele ihrer Community zu eigen machen und infolgedessen Aufgaben übernehmen.

Ubuntu Bei Ubuntu spielen das Unternehmen Canonical und ihr Gründer eine wichtige Rolle, da es Canonical-Mitarbeiter*innen Aufgaben zuweisen kann. Somit können auch Aktivitäten abgedeckt werden, die FLOSS-typisch nicht unbedingt im Fokus der Entwickler*innen liegen, beispielsweise *User Experience-Design* oder *Community-Management*. Es gibt einige *Teams*, die bestimmte Bereiche koordinieren und an deren regelmäßigen Online-Treffen User partizipieren dürfen.

Als Zuweisung von Aufgaben, die für Einsteiger*innen geeignet sind, gibt es in Ubuntu sogenannte „Papercut“ Bugs, die sehr einfach zu beheben sind und somit neue Beitragende motivieren sollen.

Neben dem üblichen Bug-Tracking werden in Ubuntu's *Launchpad* auch die Community-Strukturen abgebildet und beispielsweise auch die Übersetzung von Programm-Texten koordiniert. Es handelt sich dabei um eine Plattform, die offenbar über den üblichen Umfang eines Bugtrackers hinausgeht.

Debian In Debian sind die Pakete in der Regel Entwickler*innen zugeordnet, die damit dafür zuständig sind. Sie genießen weitgehende Autonomie, das Projekt ist somit sehr dezentral organisiert. Dennoch werden einige Aufgaben von *Delegations* übernommen und der DPL hat eine leitende Kompetenz. Neueinsteiger*innen bekommen ein Paket zugewiesen, das keinen Maintainer mehr hat. Zudem gibt es eine Kategorie „Newcomer Bugs“ mit (relativ) einfachen Aufgaben.

Arch In Arch wird lediglich der Bugtracker als Zuweisungsfunktion genannt. Hier findet also das klassische FLOSS-Modell Anwendung.

Beitragende Akteure

Jenseits der stark durch Zugriffsrechte regulierten Bereiche der Code-Repositories generieren tendenziell die Mitglieder der Community die Beiträge. Die Selektion der Mitglieder hat einen starken Einfluss auf die Beitragsbereiche. Hierbei spielen User ohne formale Rolle in weniger regulierten Beitragsarten eine größere Rolle (Feedback und Knowledge Base).

Folglich sind in Ubuntu auch Beitragende mit *Beer-Mat Knowledge* aktiv, während in Arch lediglich User mit mindestens *Primary Source Knowledge*, eher aber *Interactional Expertise* aktiv sind.

Ubuntu Offenbar sind in Ubuntu tendenziell “more less technical involved”, und zwar von *Advocacy* bis hin in den Bereich Feedback, da selbst hier Beiträge von Laien akzeptiert werden.

In der Entwicklung der Funktionalität selbst, den Code Repositories, ist die Mitarbeit aber dennoch auf Entwickler*innen mit *Contributory Expertise* beschränkt. Außerdem gibt es durch Canonical bezahlte Entwickler*innen, Designer*innen und Community-Manager*innen.

Debian *Debian Developer* und *Maintainer* spielen als Beitragende die zentrale Rolle. In Debian tragen auch User bei, haben allerdings einige technische (Tools) und organisatorische (Policies) Hürden zu überwinden und müssen sich mit den zuständigen Maintainern verständigen.

Arch Alle User übernehmen potenziell auch Entwickleraufgaben. Beitragende User haben meist zumindest *Interactional Expertise*.¹²⁶

Mitglieder, Nutzerbilder und Beitragende

Das initiale Marketing von „Ubuntu Linux for Human Beings“ wirkt weiter durch begeisterte Laien, die anderen auch einen „Umstieg“ zu Linux ermöglichen wollen. Bei Debian wird das System von Power-Usern eher unaufgeregt genutzt und geschätzt, bei Arch schließlich grenzen sich Expert*innen vom Mainstream ab, da sie sich nicht mit laienhaften Problemdiskussionen beschäftigen möchten.

Laien werden bei Ubuntu nicht nur umworben, sondern auch ihre „User Experience“ ist ein explizites Entwicklungsziel. Der Bug Report bildet hierbei eine Brücke, um auch das Feedback von Laien in die Entwicklung einzubinden und somit die Anwendbarkeit für diese Nutzergruppe zu verbessern. Mit anderen Worten wird hierbei eine Wissensdifferenz als Defizit der Entwickler*innen formuliert, die ohne entsprechendes Feedback nicht wissen können, was die Bedürfnisse der User sind. Eine Unterscheidung zwischen Entwickler*innen und Usern tritt hier aber klar zutage und es gibt sogar bezahlte Usability-Spezialisten. User mit *Interactional Expertise* vermitteln zwischen Entwicklersicht und Laiensicht über das Bug Triaging.

126 Im Bereich der Knowledge Base, insbesondere im Wiki, kann im Grunde jede*r beitragen, aber um dem Anspruch der Community zu genügen, wird eine gewisse Expertise erwartet.

Diese Bemühungen der Integration der Laien-Perspektive stehen im direkten Kontrast zur RTFM-„Policy“ bei Debian, die den Fehler bei Unverständnis der Software vor dem Bildschirm (beim User) verortet und eine Erweiterung der Dokumentation als Lösung betrachtet. Einen weiteren Kontrast bildet Arch, wo im Zweifel die Lösung im Wechsel der User zu einem einfacheren System gesehen wird.

Anforderungen

Ubuntu Die Anforderungen sind möglichst niederschwellig, und Feedback und Beiträge zur Knowledge Base können auch ohne technische Expertise mit *Beer-Mat Knowledge* erbracht werden. Für die Mitarbeit am Code ist aber (natürlich), wie bei anderen Projekten auch, *Contributory Expertise* notwendig.

Die Bedienbarkeit der technischen Tools für die Beiträge wie Bug Reporting und Übersetzung spielen dabei eine wichtige Rolle, um eben weniger technisch versierteren Usern das Beitragen möglichst einfach zu gestalten. Generell wird in Ubuntu stärker betont, dass jegliches Engagement von Bedeutung ist.

Debian Bezüglich der technischen und sozialen Organisation der Beiträge ist die Lernkurve steil. Es gibt einige Regeln, deren Beachtung rigide eingefordert wird. Es gibt aber auch niederschwelligere Beitragsmöglichkeiten über weniger regulierte Kommunikationswege, Beiträge im Wiki oder die Kommunikation mit Entwicklern im IRC.

Für das Feedback in Bug Reports ist hier *Interactional Expertise* erwünscht, im Bereich der Knowledge Base sind Beiträge basierend auf einem *Popular Understanding* denkbar.

Arch Für die Beiträge wird große Sorgfalt im Bereich von Feedback und zumindest *Interactional Expertise* erwartet. Für Beiträge zur Knowledge Base ist es denkbar, dass *Primary Source Knowledge* hier für manche Beiträge ausreichend sein kann. Das AUR verlangt darüber hinaus auch von den Usern Obacht und Sachverstand und *Contributory Expertise* wird generell von Usern erwartet.

Anhand des Vergleichs zeigen sich also starke Unterschiede der epistemischen Regime bezüglich der normativen Konfiguration und der sozialen Ordnung. Im folgenden Kapitel werde ich am Beispiel der Installationskripte der einzelnen Distributionen die normativen Einschreibungen der Software herausarbeiten, um so exemplarisch zu zeigen, wie sich die Konfigurationen der epistemischen Regime auf die Wissensprodukte auswirken. Anschließend

komme ich zurück auf die Fragestellung, um die Ergebnisse zusammenzufassen und die Fragen der Untersuchung zu beantworten.

9 Normative Inskriptionen in Software

Ich habe gezeigt, wie sich in den betrachteten Variablen der vorgestellten epistemischen Regime verschiedene Wertsetzungen widerspiegeln. Es existieren hierbei sehr unterschiedliche Grenzziehungen zwischen Usern und Entwickler*innen. Außerdem werden andere Voraussetzungen an die User gestellt bezüglich ihrer Expertise – sowohl für die Nutzung der Software, als auch für den Beitrag zur Community.

Diese spezifischen Wertsetzungen und Grenzziehungen fließen auch in die gemeinsamen Wissensprodukte ein. In Bezug auf die Knowledge Bases der Communities und die Feedback-Möglichkeiten wurde deutlich, dass ein gewisses Verständnis notwendig ist und das vorausgesetzte Wissen für die Beiträge sowohl die Dokumentation als auch die Gestaltung der Software prägt, beispielsweise in der Berücksichtigung der Verständlichkeit von Benutzerfragen sowie bei der Gestaltung von Benutzerführung und Design. Um die Wirkungen der epistemischen Regime nun noch deutlicher herauszuarbeiten und zu zeigen, wie die Wertsetzungen der epistemischen Regime sich in Software materialisieren, gehe ich in diesem Kapitel der zweiten Forschungsfrage nach: „Wie wirken Regime auf das gemeinsame Wissensprodukt?“ Dafür werde ich nun exemplarisch ein Software-Programm jeder Fallstudie als Wissensprodukt analysieren und diese miteinander vergleichen.

Da für die Nutzung von Linux bislang kaum ein User auf ein vorinstalliertes System zurückgreifen kann, muss dieses vor der Benutzung auf dem Computer installiert werden. Das Installationsprogramm ist somit die erste Software der Community, mit der User in Kontakt kommen. Darum bietet sich dieses Programm besonders für den Vergleich der Installationsroutinen der Communities an.¹²⁷

Ein weiterer Grund für die Auswahl ist, dass sich die installierten Anwendungsprogramme auf den Systemen sich nicht notwendigerweise unterscheiden, da die meisten Anwendungs-Programme aus dem Upstream kommen. Unterschiede ergeben sich vor allem in den Versionen der Programme, in der Ausgangskonfiguration (den Voreinstellungen) und in der Administration des

127 Es gibt bei jedem System verschiedene Installationswege; ich greife hier die klassischen und von der Community empfohlenen Installationsprogramme heraus, um die damit verbundenen Setzungen herauszustellen.

Systems. Hierfür schreiben die Communities unterschiedliche Werkzeuge. Das grundlegendste Werkzeug ist jedoch das Installations-Skript. Es zeigt sich außerdem, dass der Installer auch eine soziale Funktion der Selektion potenzieller User erfüllt und dadurch das Nutzerbild der Community reproduziert.

Im Folgenden beschreibe ich zunächst, wie die Installationsdatei über die Hauptseite der jeweiligen Distribution bezogen wird. Dies stellt gewissermaßen den Erstkontakt mit der Selbstdarstellung der Community dar. Die Erstellung des Installationsmediums überspringe ich, da es hierfür viele Methoden gibt, die nicht distributionsspezifisch sind. Anschließend beschreibe ich die einzelnen Schritte der Installations-Skripte und analysiere die impliziten Anforderungen an die User anhand der vorgeschlagenen und zur Wahl gestellten Optionen und der im Programm angezeigten Beschreibungstexte.

Im Fall von Arch handelt es sich im Grunde weniger um ein Installations-Skript im Sinne eines automatisierten Ablaufs, vielmehr ist das Skript hier eine Textdatei mit Installationshinweisen, die neben der Kommandozeile zum Ausführen der entsprechenden Befehle auf dem Installationsmedium zur Verfügung gestellt wird. Der User hat also gegenüber den vorgegebenen Handlungsmöglichkeiten der Installationsroutinen in den anderen Fällen maximale Gestaltungsmöglichkeiten. Hier materialisiert sich auf konsequente Weise der Anspruch an die User, ihr System selbst zu gestalten und sich dafür das notwendige Wissen anzueignen – der User wird durch die Befolgung der Hinweise selbst zum Installations-Skript.

Hieran wird deutlich, dass die Benutzerführung eines Programms sehr unterschiedlich gestaltet werden kann und dass in der Gestaltung sehr starke Festlegungen enthalten sind. Diese wiederum determinieren die Handlungsmöglichkeiten der User auf verschiedenen Ebenen: durch Informationen über zu wählende Optionen, durch die Gestaltung der Dialoge, die Voreinstellungen und die visuelle Darstellung und schließlich durch das Bereitstellen oder Verbergen bestimmter Auswahlmöglichkeiten.

Zur Schärfung der Konturen des Vergleichs der Installations-Skripte verwende ich einen Vergleichsrahmen für die beeinflussende Wirkung von Technik, den ich in den letzten Jahren zusammen mit dem *Berlin Script Collective*¹²⁸ 2017 erarbeitet habe. Es handelt sich dabei um eine Weiterentwicklung und Ausdifferenzierung des Skript-Begriffs von Akrich (1992), die da-

128 Neben mir sind das in alphabetischer Reihenfolge Jochen Gläser, Grit Laudel, Martin Meister, Fabia Schäufole, Cornelius Schubert und Ulla Tschida.

mit normative Inskriptionen in Technik beschreibt. Der Begriff des Skriptes verweist hierbei auf die Regie-Anweisungen in einem Film-Skript und versucht, die von den Entwickler*innen vorgesehene projizierte Nutzung der Technik durch die User zu fassen. Der Vergleichsrahmen wurde entwickelt, um verschiedene Technologien im Hinblick auf ihre Wirkung auf die Handlungsabläufe ihrer User vergleichend zu analysieren. Die verschiedenen Arten des Einflusses differenzieren sich hierbei insbesondere in materialisierte, zwingende Vorgaben, Anreizstrukturen und Informationen. In der unterschiedlichen Gestaltung der Benutzerführungen finden sich in den Installationsskripten genau diese Aspekte wieder. Die Eingrenzung von Handlungsmöglichkeiten als Komplexitätsreduktion für nicht technische User steht dabei im Gegensatz zur völligen Wahlfreiheit technisch versierter User, denn die maximalen Handlungsspielräume gehen mit der Notwendigkeit einher, das Wissen für bestimmte Entscheidungen zu haben – oder zu erwerben. Da die Freiheitsgrade der Nutzung und die Gestaltung der Benutzerführung im Fokus des Vergleichsrahmens stehen, eignet er sich für die hier durchzuführende Analyse.

Vergleichsdimensionen für die Skriptanalyse

In aller Kürze möchte ich die Dimensionen des Analyserahmens skizzieren – die Anwendung im weiteren Verlauf des Kapitels veranschaulicht die konkrete Bedeutung.

- *Adressierung und Anforderungen:* Eingeschriebene Definition der Nutzergruppe, Zugänglichkeit der Technologie für verschiedene Nutzer*innen. Für welche User ist die Technik zugänglich bzw. welche Anforderungen sind impliziert?
- *Spezifität des Nutzungszwecks:* Anwendungsbreite und Zielsetzung der Technik.
- *Art und Stärke des Einflusses:* Wie stark ist die Einflussnahme auf den User, wird er *gezwungen*, werden bestimmte Optionen vorgegeben (*Ausstattung*) oder ihm *Anreize* gesetzt oder wird ihm durch *Informationen* eine bestimmte Interpretation der Situation und damit eine Handlung nahegelegt?
- *Verteilung der Handlungskontrolle und Flexibilität der Nutzung:* Liegt die Kontrolle eher auf der Seite des Users oder des Artefakts?
- *Homogenität der Kontrollverteilung im Handlungsablauf:* Gibt es wechselnde Phasen zwischen User-Eingabe und Technikablauf?

- *Materielle Gegebenheiten des Kontextes:* Welche Rolle spielen die Anwendungssituation und die größeren Zusammenhänge, in die das Skript eingebettet ist?
- *Sichtbarkeit des Skripts:* Wie gut sind das Skript und die dahinter liegenden Intentionen sichtbar?

In den Fallbeschreibungen werde ich die Ergebnisse punktuell den genannten Dimensionen zuordnen, um sie anschließend in Tabellen zusammenzufassen und abschließend vergleichend gegenüberzustellen.

9.1 Ubuntu Linux: In wenigen Schritten zum Betriebssystem

Bei Ubuntu fällt auf, dass die Installation üblicherweise von einem sogenannten Live-Medium erfolgt. Das ist ein USB-Stick, eine CD oder DVD,¹²⁹ von wo aus man ein umfangreiches vorinstalliertes Desktop-System (mit grafischer Oberfläche und den üblichen Büro- und Internet-Anwendungen) erst niederschwellig testen und anschließend direkt auf den Computer installieren kann. Dies ermöglicht Einsteiger*innen, sich zunächst von der Funktionalität und Attraktivität des Systems zu überzeugen, denn Einsteiger*innen und Neulinge sind aufgrund des Nischendaseins von Linux im Bereich der Desktop-PCs im Grunde immer Umsteiger*innen. User können sich daher mithilfe des Live-Mediums von der Nutzbarkeit des Systems überzeugen.¹³⁰

Das grafische Design springt dabei sofort ins Auge und hebt sich in seiner Ästhetik von den beiden anderen Fällen deutlich ab. Das Installationsprogramm weist gleich zu Beginn darauf hin, dass es von Vorteil ist, wenn das Notebook während der Installation am Strom angeschlossen ist. Hier wird schnell klar, dass der Laie auf einem recht niederschweligen Wissensstand abgeholt wird und befähigt werden soll, ein Betriebssystem zu installieren.

129 Üblicherweise wird zunächst eine Image-Datei heruntergeladen und daraus das Medium erstellt – entweder durch Brennen einer CD/DVD oder durch das Kopieren auf einen USB-Stick. Häufig werden aber auch vorbereitete DVDs angeboten, beispielsweise in Computer-Zeitschriften.

130 Vergleiche auch die explizite Vision, Microsoft die Hegemonie streitig zu machen (S. 109).

Hier zeigt sich schon deutlich, dass dem User die Ubuntu-Installation möglichst angenehm und unkompliziert gemacht werden soll. Ein einfacher User, der eine DVD in die Hand gedrückt bekommt, soll sich möglichst angesprochen fühlen vom Design der Oberfläche und der niederschweligen Benutzerführung.

9.1.1 Download: „Help shape the future of Ubuntu“

Geht man auf ubuntu.com im Bereich *Desktop* auf *Download Ubuntu*, kann man verschiedene Ubuntu-Versionen wählen. Die erste angegebene und empfohlene Variante ist die sogenannte LTS-Version. LTS, „Long Term Support“, bedeutet, dass über fünf Jahre Sicherheits-Updates zur Verfügung gestellt werden. In diesem Zeitraum werden zwar wichtige Sicherheitslücken behoben, aber darüber hinaus werden keine großen Veränderungen angewendet, auf die man sich als User einstellen müsste oder die die Kompatibilität zwischen Programmen tangieren.

Folgt man dem Link, gelangt man auf eine Seite mit dem Titel „Contribute to Ubuntu“ (Abb. 9.1; S. 216). Dort kann man unter der Überschrift „Help shape the future of Ubuntu“ mithilfe eines Reglers einstellen, wie wichtig bestimmte Bereiche persönlich bewertet werden und wie stark diese von Ubuntu künftig entwickelt werden sollen. Die Gewichtung erfolgt in US-Dollar. Die Bereiche betreffen Konvergenz von Desktop-PCs und mobilen Geräten, Cloud Computing, Internet der Dinge, Community-Projekte. Zuletzt kann ein „Trinkgeld“ für Canonical ausgegeben werden. Der sich aus der Summe der Bewertungen ergebende Wert wird mit dem Preis eines materiellen Produktes verglichen, etwa einem Bier (5 US-\$), einem T-Shirt, eine Jeans oder sogar einem Kamel (1000 US-\$). Die Voreinstellung liegt beim Preis einer „King Kong versus Godzilla“-DVD für 15 US-\$. Neben dem prominent sichtbaren Bezahlknopf gelangt man zum Download ebenfalls über einen kostenfreien Link mit der Bezeichnung „Not now, take me to the download“. Hier wird der an sich abstrakte Wert eines Betriebssystems in Relation gestellt zu konkreten, fassbaren Produkten, um dem User zu veranschaulichen, wofür er bereitwillig Geld ausgibt, und ihn so zu animieren, auch einen Obolus für das System zu bezahlen. Außerdem wird die Möglichkeit suggeriert, durch eine Aufteilung des Geldes eine Stimme abzugeben über die eigenen Prioritäten und Interessen.

Help shape the future of Ubuntu

Tell us what is most important to you.

Ubuntu Desktop
Make the desktop even more amazing.

\$ 3

Ubuntu for cloud computing
I want Ubuntu running my cloud and as a guest in my cloud of choice.

\$ 3

Ubuntu for things
I want a secure, upgradeable Internet of Things, powered by Ubuntu.


\$ 3

Community projects
I support LoCo teams, UbuCons and other events, upstream projects and all the good work the community does.

\$ 3

Tip to Canonical
Hats off for making Ubuntu possible. Keep it up.

\$ 3



The same price as
King Kong versus Godzilla on DVD
\$15

Your contribution
\$ 15

[Not now, take me to the download >](#) [Pay with PayPal](#)

Abb. 9.1 Download-Seite von Ubuntu

Die Bereitstellung von Informationen (Einflussmodus) konzentriert sich zunächst also darauf, dem User den Wert des *freien* Produkts nahezubringen, um ihn zu einem ersten Beitrag auf monetäre Art zu bewegen. Dadurch wird auf ein arbeitsteiliges Verhältnis der Konsumption und Produktion, zwischen Hersteller*in und Verbraucher*in hingewiesen, was bei FLOSS ansonsten nicht selbstverständlich ist, da Verbraucher*innen grundsätzlich auch potenzielle Hersteller*innen sind.

Die Datei ist 1,4 Gigabyte (GB) groß und während des Downloads¹³¹ werden Links angezeigt zum „Installation Guide“, zu einer Seite, die Ubuntu kommerziellen Support bewirbt sowie zur Frage-und-Antwort-Plattform „Ask Ubuntu“. Neben dem Hinweis auf eine Anleitung und den Support der Community erfolgt hier also ein direkter Hinweis auf den individuellen Produktsupport als Dienstleistung durch Canonical.

9.1.2 Ubuntu: Ausprobieren und installieren

Startet man die Ubuntu-DVD, erscheinen zunächst neben der Auswahl der Sprache zwei Piktogramme, die „Ubuntu ausprobieren“ oder „Ubuntu installieren“ als Optionen anbieten (Abb. 9.2; S. 218).

Wird die erste Option „Ubuntu ausprobieren“ gewählt, startet ein sogenanntes Live-System, ein Betriebssystem, das direkt vom Medium läuft und Ubuntu als fertiges System erfahrbar macht, ohne zunächst die Konfiguration des Computers zu verändern.

Hier können also das „Look and Feel“ des Systems ganz einfach ausprobiert und die Standard-Applikationen wie Office-Anwendungen und Internet-Browser direkt benutzt werden. Auf dem Desktop des Live-Systems findet sich ein Installationsknopf, durch den ebenfalls der eigentliche Installationsprozess gestartet werden kann. Dieser kann auch vom Startbildschirm des Live-System aus gestartet werden – nur, dass dann im Hintergrund bereits ein grafisches System läuft, das auch während der Installation genutzt werden kann (zum Beispiel, um im Internet zu surfen).

¹³¹ Die von den Vergleichsfällen empfohlenen Download-Wege über *BitTorrent* o.Ä. sind hier nur unauffällig unter einem großen Download-Button genannt.

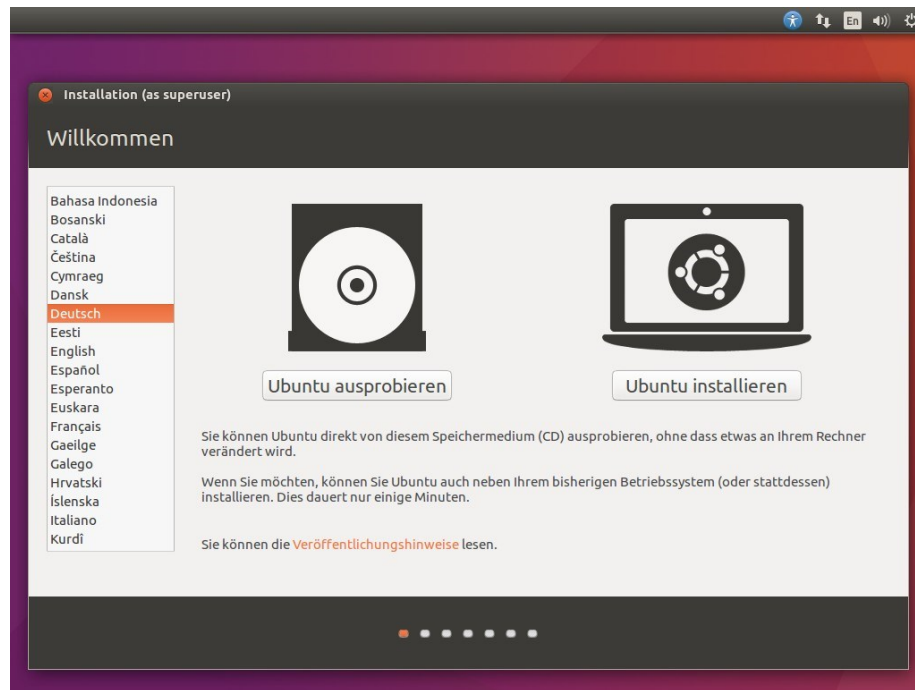


Abb. 9.2 Startbildschirm des Ubuntu-Installationsmediums

Hier zeigt sich eine *Adressierung* von Desktop-Usern, die hier ihre ersten Erfahrungen mit Linux machen und sich das System zunächst anschauen wollen. Linux-Einsteiger*innen sind in aller Regel eben auch Umsteiger*innen von anderen Systemen und bekommen durch das Ausprobieren im besten Fall durch die Präsentation eines schönen und funktionierenden Systems einen *Anreiz*, die Installation tatsächlich durchzuführen. Ein erfahrener Linux-User hat weniger Interesse, ein Desktop-System „auszuprobieren“, da die einzelnen Anwendungen sich im Grunde nicht wesentlich unterscheiden. Die spannenderen Unterschiede zeigen sich bei der Konfiguration des Systems und bei den spezifischen Einstellungen des Systems, die sich im Live-System weniger ausgiebig testen lassen.

Im nächsten Schritt (Abb. 9.3; S. 219) wird darauf hingewiesen, dass für „beste Ergebnisse“ der Installation sichergestellt werden sollte, dass eine gewisse Menge an Festplatten-Speicher vorhanden ist, der Computer (hier: ein Notebook) am Strom angeschlossen ist, und eine Internet-Verbindung besteht. Dies erinnert den User an sehr basale Notwendigkeiten und setzt die Wissens-Anforderungen somit äußerst gering an (*Adressierung von Usern*).

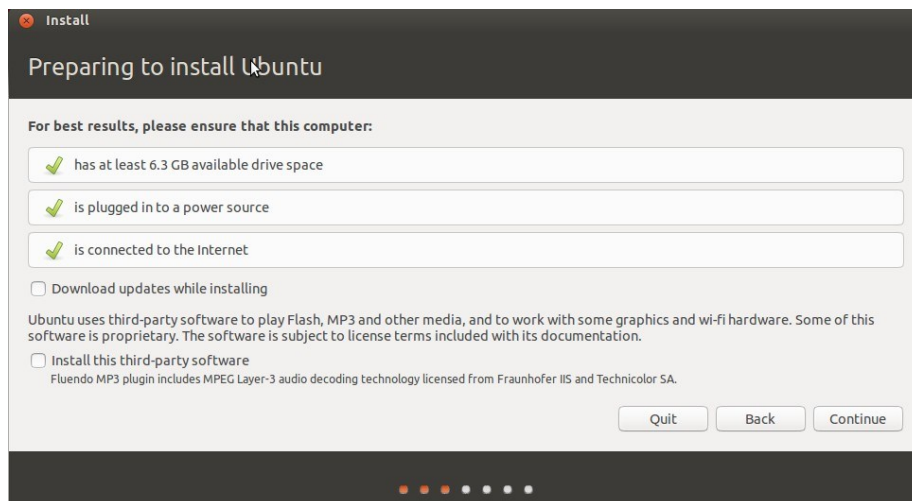


Abb. 9.3 Vorbereitung der Ubuntu-Installation

Dazu sind zwei Optionen wählbar: zum einen, dass während der Installation aktuellere Versionen installiert werden, als auf dem Medium enthaltenen, und zum anderen, sogenannte „Third Party“-Software zu installieren. Dies bezieht sich vor allem auf proprietäre Treiber, die für manche Hardware benötigt wird, und auf proprietäre Medienformate wie MP3 und Flash, die für das Abspielen oder Generieren von bestimmten Medien-Dateien notwendig sind. Hier wird dem User also auch sehr niederschwellig angeboten, Treiber zu nutzen, die nicht aus dem FLOSS-Pool bezogen werden können – anders als bei Debian und Arch ist hierfür kein weiteres Vorwissen nötig. Ist das Häkchen gesetzt, wird diese Software automatisch installiert, während bei den anderen beiden Distributionen hier Handarbeit notwendig ist.

Im nächsten Schritt (Abb. 9.4; S. 220) bietet der Installer, abhängig von der aktuellen Konfiguration des Computers, einen Vorschlag an, wie die Festplattenkonfiguration verändert werden sollte, um Ubuntu zu installieren. Denn für die Installation muss ein separater Bereich, eine Partition, auf der Festplatte eingerichtet werden, auf dem das System gespeichert wird. Die Vorauswahl der Software bedarf keiner näheren Beschäftigung mit der Frage der Partitionierung – wie etwa, welches Dateisystem dafür verwendet werden soll. Darüber hinaus werden noch zwei Zusatzoptionen angeboten, die über das einfache Setzen eines Häkchens ebenfalls automatisiert durchgeführt werden. Diese Optionen werden in einem Satz (der auch auf einen Bierdeckel passen würde) näher erklärt (*Beermat Knowledge*). Wer doch selber Hand

anlegen will und eigene Vorstellungen hat, wie die Festplatte einzurichten sei, kann die Alternativ-Option „Something else“ auswählen. Hier wird der User zum Partitionierungs-Programm geführt und kann die Partitionierung im Detail selbst durchführen, wofür aber etwas mehr Wissen benötigt wird. Dem User wird hier ein klarer *Anreiz* gegeben, die Voreinstellung zu übernehmen, denn er will ja das System installieren und nicht „etwas anderes“ machen. Die Informationen sind an dieser Stelle recht knapp und einfach gehalten. Wer den vorgegebenen Standardpfad verlässt, sollte wissen, was er oder sie tut.

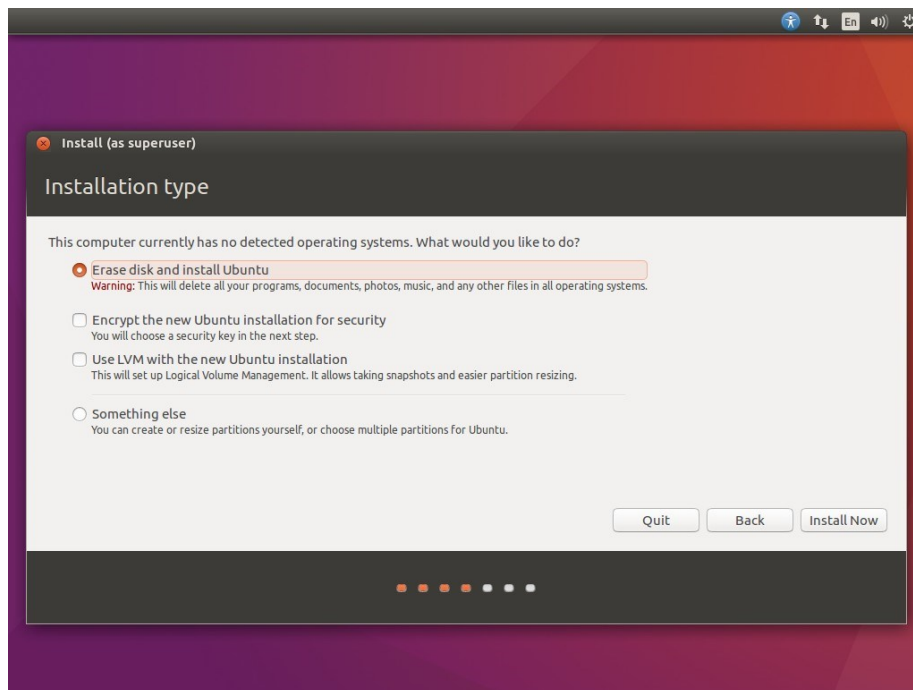


Abb. 9.4 Vorbereitung der Festplatte (Ubuntu)

Nach der automatischen (oder manuellen) Festplattenkonfiguration kann im nächsten Schritt die Installation gestartet werden. Dafür muss die vorgeschlagene Konfiguration final bestätigt werden (Abb. 9.5; S. 221), um den Installationsprozess in Gang zu setzen. Dieser „Zwang“ ergibt sich aus der Notwendigkeit, dass ohne die Entscheidung, jetzt auf die Festplatte zu schreiben, auch kein System installiert werden kann, da hierzu Dateien für die spätere Verwendung kopiert werden müssen. Dieser Schritt tritt hier bemerkenswerterweise recht früh ein, im Grunde als erstes, somit ist die wich-

tigste Entscheidung bereits erledigt: der Schreibzugriff auf die Festplatte und damit die Entscheidung, das System zu installieren. Alle weiteren Konfigurationen werden parallel zum Kopiervorgang vorgenommen. Hier finden also genau genommen zwei Schritte parallel statt, ein automatischer (Kopiervorgang) und ein manueller (Eingabe von Konfigurationsparametern). Diese Anordnung der Schritte impliziert eine gewisse Anreizstruktur: Die Installation läuft bereits, die Entscheidung für das neue System ist während der weiteren Konfiguration schon gefällt, ein Zurück gibt es an dieser Stelle im Grunde nicht mehr („Zwang“).

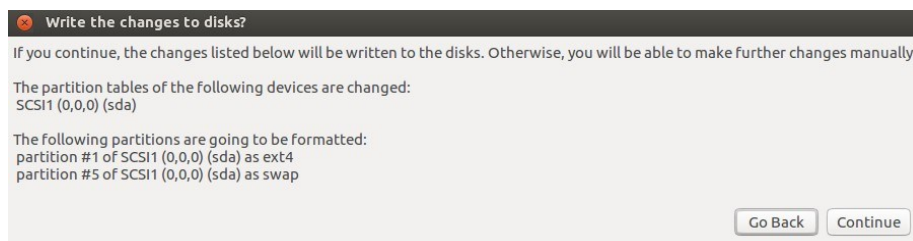


Abb. 9.5 Bestätigung der Festplattenkonfiguration (Ubuntu)

Der Installationsprozess kann daher in zwei Phasen eingeteilt werden: eine erste, in der die wichtigsten Parameter eingestellt werden, um das System zu kopieren, und eine zweite, in der, parallel zum hintergründigen Vorgang des Kopierens der notwendigen Dateien auf die Festplatte, noch weitere Einstellungen vorgenommen werden. Dadurch ergibt sich ein *Point of no return*, der vor Ende der finalen Konfiguration eintritt und tendenziell verhindert, dass im späteren Verlauf die Installation vom User abgebrochen wird.

Zu den anschließenden Einstellungen zählen Zeitzone, Tastatur und die Einrichtung eines Nutzerkontos. Die Lokalisierung ist dabei grafisch anschaulich dargestellt. Eine Kenntnis der Zeitzonen ist dafür nicht nötig; es reicht, den richtigen Fleck auf der Weltkarte auszuwählen (Abb. 9.6; S. 222).

Bei der Einstellung des Tastatur-Layouts besteht auch die Möglichkeit, die passende Tastatur *automatisch* zu ermitteln – ein besonders starker Kontrast zur unten beschriebenen Konfiguration in Arch. Dafür wird der User vom Software-Assistenten aufgefordert, bestimmte Tasten zu drücken, um über die User-Eingabe das richtige Keyboard zu ermitteln. Es ist also nicht notwendig, die verschiedenen Modelle und Möglichkeiten zu kennen, um die richtige Option zu wählen.



Abb. 9.6 Lokalisierung des System (Ubuntu)

Im Vergleich zu den anderen Installern kommt diese Konfiguration sehr spät, da die bisherigen Einstellungen auch per Maus eingegeben werden konnten – selbst die manuelle Festplattenpartitionierung kommt ohne Tastatureingaben aus. Nun steht aber die Einrichtung des User-Accounts an; dazu sollen Name, Nutzer-Kennung und Passwort eingegeben werden (Abb. 9.7; S. 223).

Im Gegensatz zu den anderen Fällen gibt es hier lediglich den User-Account, obwohl Unix-Systeme traditionell einen separaten Systemadministrator-Account implizieren. Dieser wurde bei Ubuntu von Anfang an versteckt, und die entsprechende Funktionalität in den User-Account eingebaut. Im Gegensatz zu Computern, die von separaten Administrator*innen betreut werden, ist dieses Vorgehen für einen Einzelplatz-Rechner praktikabel und für Umsteiger*innen von Microsoft-Systemen deutlich einfacher nachzuvollziehen – die konsequente Trennung vom Administrator-Konto war bis zur Einführung von Ubuntu in Microsoft-Windows-Systemen nicht die Regel.¹³²

132 Aus Sicherheitsgründen erscheint ein separates Administrationskonto allerdings sehr sinnvoll, eine Absicherung über eine separate Passwordeingabe für systemrelevante Änderungen wurde aber anderweitig eingebaut.

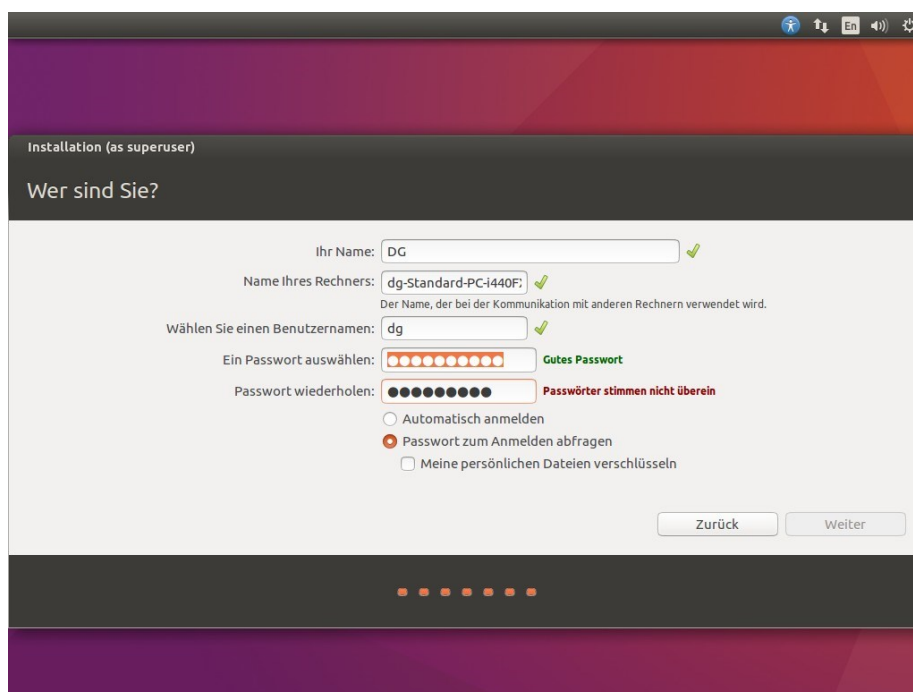


Abb. 9.7 User einrichten (Ubuntu)

Schließlich wird bei der Account-Erstellung ein Rechnername vorgeschlagen – eine Einstellungsoption, die im Grunde nur in lokalen Netzwerken von Bedeutung ist, hier aber einfach im Zuge der Namensfrage des Users erfolgt und somit für Laien sinnvoll kontextualisiert ist. Bei der Passwort-Erstellung erhält der User visuelles Feedback über die Stärke des Passworts. Optional kann der User eine automatische Anmeldung wählen, um nicht bei jedem Start des Computers seine Anmeldedaten eingeben zu müssen. Darüber hinaus kann der User eine Verschlüsselung der Account-bezogenen Daten per Mausklick automatisch einrichten. Der User bekommt also eine starke Rückmeldung über seine Eingaben (sogar in Farbe) und kann auch erweiterte Funktionalitäten wie Verschlüsselung per Klick hinzufügen.

Mit diesem Schritt ist die Konfiguration vollständig und während der Computer die restlichen Dateien auf dem Rechner installiert, wird dem User schon ein Vorgeschmack auf die installierten Programme gegeben, ferner Hinweise, wie beispielsweise neue Programme gefunden und installiert werden können. Dies ist beispielsweise bei Microsoft schon lange üblich und gibt dem User einen Ausblick, was ihn nach dem Start des neuen Systems

erwartet. In den beiden anderen Fällen ist diese Art von Vorschau nicht in den Installer eingebaut.

Nachdem die Installation abgeschlossen ist, wird der User schließlich darauf hingewiesen, vor dem Neustart das Installationsmedium zu entfernen, damit das neu installierte System von der Festplatte gestartet wird und nicht wieder die Installations-CD geladen wird.

9.1.3 Inskription des Ubuntu-Installers

Die Inskriptionen nach dem beschriebenen Analyseraster finden sich in Tabelle 9.1.

Tab. 9.1: Zusammenfassung der Inskriptionen des Ubuntu-Installers

Dimension	Fall Ubuntu
Adressierung/Voraussetzung von Usern	Ein- und Umsteiger mit <i>Beer-Mat Knowledge</i>
Spezifität des Zwecks	hoch: Installation eines Arbeitsplatzrechners (Desktop-PC)
Art des Einflusses	Anreiz durch Live-System und schöne Grafik; klare, begrenzte Voreinstellungen (<i>Ausstattung</i>); früher <i>Point of no return</i> und starke Benutzerführung (<i>Zwang</i>)
Verteilung der Kontrolle	weitestgehend liegt die Kontrolle beim Skript
Homogenität der Kontrollverteilung	homogen, relativ wenige Eingabephasen
materielle Gegebenheiten des Kontextes	unabhängig von einer Internet-Verbindung
Sichtbarkeit des Skripts	Parallelität von Konfiguration und Kopiervorgang (der im Hintergrund verläuft)

Die *Adressierung* des Installationsmediums richtet sich an Einsteiger*innen und Umsteiger*innen von anderen Systemen, die über ein grobes schematisches Verständnis von Computern verfügen (*Beer-Mat Knowledge*).

Dies zeigt sich auch in der Ausrichtung des *Zwecks* des Mediums, das sehr *spezifisch* einen üblichen Arbeitsplatzrechner (Desktop-PC) mit allen dazugehörigen Programmen vorinstalliert. Das impliziert eine große Menge an Software, die als für diesen Zweck sinnvoll erachtet wird (und schließt Software für andere Zwecke aus, beispielsweise für den Serverbetrieb relevante Programmpakete).

Art des Einflusses: Das enthaltene Live-System *informiert* den User über die Beschaffenheit des Systems und gibt über die konkrete Erfahrung bestenfalls einen *Anreiz*, das System zu installieren. Die Vorgabe von wenigen Optionen strukturiert in den einzelnen Installationsschritten das Ergebnis (*Ausstattung*). Die Installation verlangt zu einem frühen Zeitpunkt den Schreibzugriff auf die Festplatte und fängt dann unwiderruflich mit dem Kopiervorgang an („Zwang“). Die Benutzerführung hat eine zwingende Reihenfolge als Struktur. Die *Verteilung der Kontrolle* liegt sehr stark beim Skript; der User hat eine geringe Flexibilität. Nach der anfänglichen Konfiguration der Festplatte wird der Kopiervorgang gestartet und parallel zur automatischen Installation des Systems darf der User einige wenige Eingaben und Entscheidungen treffen.

Die *Kontrollverteilung* liegt relativ *homogen* beim Skript.

Materielle Gegebenheiten des Skriptes: Das Medium enthält – im Gegensatz zu den anderen betrachteten Fällen – alle notwendigen Dateien auf dem Installationsmedium, und ist somit auch ohne Internet-Verbindung erfolgreich nutzbar.

Die *Sichtbarkeit des Skripts* verändert sich mit der Konfiguration der Festplatte: Zunächst kann der User sich im Skript bewegen und geht schrittweise vor, anschließend wird im Hintergrund das System installiert, während der User noch einige Einstellungen macht. Sind diese abgeschlossen, wird aber durch einen Fortschrittsbalken der weitere Installationsprozess angezeigt.

9.2 Debian Linux: Zwischen Anfänger- und Expertenoptionen

Die Installations-CD enthält im Standard-Umfang kein Live-System. Wie oben ausgeführt, ist dies für einen erfahreneren Linux-User auch weniger interessant.¹³³ Das Design ist deutlich schlichter und funktionaler als beim Ubuntu-Installer und es gibt mehr Konfigurationsmöglichkeiten, beispiels-

¹³³ Und für die Wartung eines beschädigten Systems (das nicht mehr startet) lassen sich die meisten Installations-CDs auch als reduziertes Live-Medium nutzen – also ohne grafische Oberfläche –, wenn man weiß wie.

weise auch die Wahl zwischen verschiedenen Standard-Konfigurationen je nach Einsatzzweck des Computers (z.B. Server-Betrieb).

9.2.1 How to get the Debian installer

Auf der Website des Debian-Projektes werden verschiedene Wege des Downloads angeboten.¹³⁴ Wählt man einen klassischen Download über `http`¹³⁵, wird man darauf hingewiesen, möglichst nicht den Internet-Browser für den Download zu verwenden, sondern Programme, die abgebrochene Downloads wieder aufnehmen können. Gerade bei großen Dateien wie den Installationsmedien ist die Gefahr, dass der Datentransfer abbricht, relativ hoch, und in diesem Fall ist es sinnvoll, wenn die schon heruntergeladenen Daten erhalten bleiben und der Download nicht von Neuem beginnen muss.¹³⁶

Neben der Wahl des Download-Weges wird dem User die Wahl zwischen einem stabileren, aber weniger aktuellen System („stable“) und einem aktuelleren, aber weniger stabilen System („testing“) angeboten, außerdem finden sich zahlreiche CDs, aber auch eine DVD mit Live-System, sowie ein minimales „Net-Install“-Medium. Auffallend ist außerdem, dass das funktionale, puristische Design der Website sehr textlastig und einfach gestaltet ist, also eher auf die funktionale Information ausgerichtet ist, statt ein „schönes“, ansprechendes Design zu präsentieren.

Zwischen den verschiedenen Dateien auf der englischen Download-Seite werden die Unterschiede zwischen den Images erklärt und schließlich wird

134 Insbesondere wird dem User nahegelegt, eine Netinstall-Version herunterzuladen, da diese sehr klein ist und dann nur die benötigten Dateien direkt aus dem Netz geladen werden. Eine weitere Empfehlung ist die Verwendung von Jigdo, was aber zusätzliche Software zum Download erfordert. Ebenso wird der Download via *BitTorrent* vorgeschlagen, da sich hier die Netzwerklast auf viele User verteilt, die die entsprechende Datei auf ihrem jeweiligen Rechner haben. Für die Beschreibung hier habe ich den Weg des ansonsten verbreiteten `http`-Downloads verwendet. Näheres zu den Download-Optionen siehe <https://www.debian.org/CD/>, letzter Aufruf 15.4.2017.

135 <https://www.debian.org/CD/http-ftp/>, letzter Aufruf 15.4.2017

136 Die Debian Community versucht hier für eine ressourcenschonende Nutzung des Internets zu motivieren. Die meisten User sind sich nicht darüber im Klaren, welche Kosten mit dem Betrieb eines Download-Servers verbunden sind – abhängig vom notwendigen Download-Volumen, das über entsprechende Vorgehensweisen reduziert werden kann.

auch eine inoffizielle „non-free“-Firmware-Variante angeboten. Diese enthält – ähnlich wie Ubuntu – auch proprietäre Treiber und erleichtert so die Installation für vielerlei Hardware erheblich. Insbesondere ermöglicht dies die niederschwellige Nutzung von Netzwerkkarten, für die es keine FLOSS-Treiber gibt. Andernfalls müssen proprietäre Treiber während des Installationsvorgangs über lokale Medien eingebunden werden. Dazu benötigt man gegebenenfalls einen zweiten Rechner mit Internet-Zugang und muss wissen, wo man diese Treiber findet und wie man sie entsprechend in den Installer einbindet.¹³⁷

Die Website stellt also allerlei Download-Optionen bereit und fokussiert einen guten Teil der bereitgestellten Informationen darauf, einen effizienten Download-Weg zu wählen, der dem Laien-User unbekannt sein dürfte (*Art des Einflusses: Information*). Damit implizieren die Informationen ein gewisses Verständnis der beschriebenen Wahloptionen oder zumindest die Bereitschaft, sich damit auseinanderzusetzen, um eine informierte Entscheidung über den Download-Weg und das richtige Medium zu treffen. Die Adressierung wendet sich also eher an User, die ein gewisses technisches Interesse mitbringen. Die Datei für die offizielle Installations-CD ist 630 Megabyte groß, während die inoffizielle deutlich größer ist und gut über 1 Gigabyte benötigt.¹³⁸

9.2.2 Debian Installer

Nach Starten der Installations-CD kann man auswählen zwischen Installation, grafischer Installation, erweiterten Optionen, einer Hilfefunktion und einem sprach-unterstützten Installationsmodus (Abb. 9.8; S. 228). Die Grafik ist sehr funktional aufs Wesentliche reduziert, Textboxen werden mit Rechtecken und einfachen Strichen dargestellt, was in der Ästhetik etwas an frühere DOS-Fenster erinnert – und im Sinne der Beeinflussung vergleichsweise geringe *Anreize* durch eine hübsche Darstellung bietet. Es gibt auch eine Variante mit einer Grafik in höherer Auflösung, die aber im ihrem schlichten

137 Die inoffizielle *non-free*-Variante beinhaltet – wie Ubuntu – auch ein Live-System, aus dem der Installer gestartet werden kann.

138 Aus historischen Gründen ist die Installations-CD die CD Nummer eins, die den Installer enthält. Viele weitere CDs enthalten unzählige Programme, die aber heute üblicherweise bei Bedarf direkt über das Internet nachgeladen werden.

Aufbau dem textbasierten Installer weitgehend gleich ist¹³⁹ und sich immer noch deutlich von der bunten Grafik des Ubuntu Installers unterscheidet.

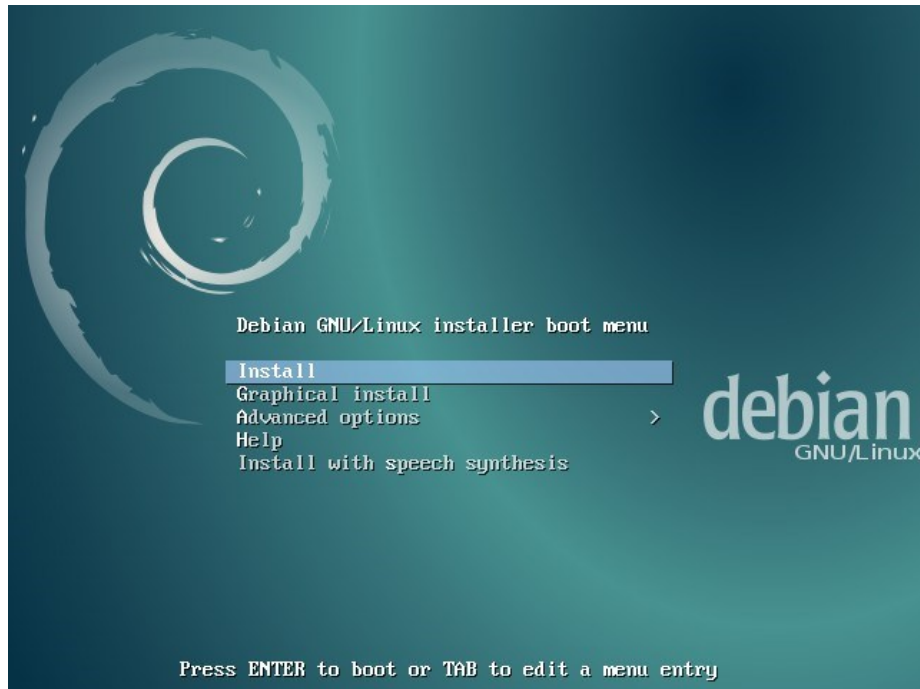


Abb. 9.8 Startbildschirm Debian-Installer

Anschließend wird man nach Sprache, Land und Tastatur-Layout gefragt (Abb. 9.9; S. 229). Das Tastatur-Layout wird hier zu einem sehr frühen Zeitpunkt abgefragt, da zu Beginn schon Texteingaben notwendig werden. Die Auswahl erfolgt über eine Liste von Optionen und benötigt keine Texteingabe.

Als nächstes wird das Netzwerk automatisch konfiguriert (wenn keine proprietären Treiber für die Netzwerkkarte notwendig sind). Dafür sind die ersten Texteingaben des Users notwendig, denn *Rechnername* und *Domain-Name* müssen an dieser Stelle eingegeben werden (Abb. 9.10; S. 229).

139 Siehe auch <https://www.debian.org/releases/stable/amd64/cho6s01.html.de#gtk-using>, letzter Aufruf 3.1.2017.

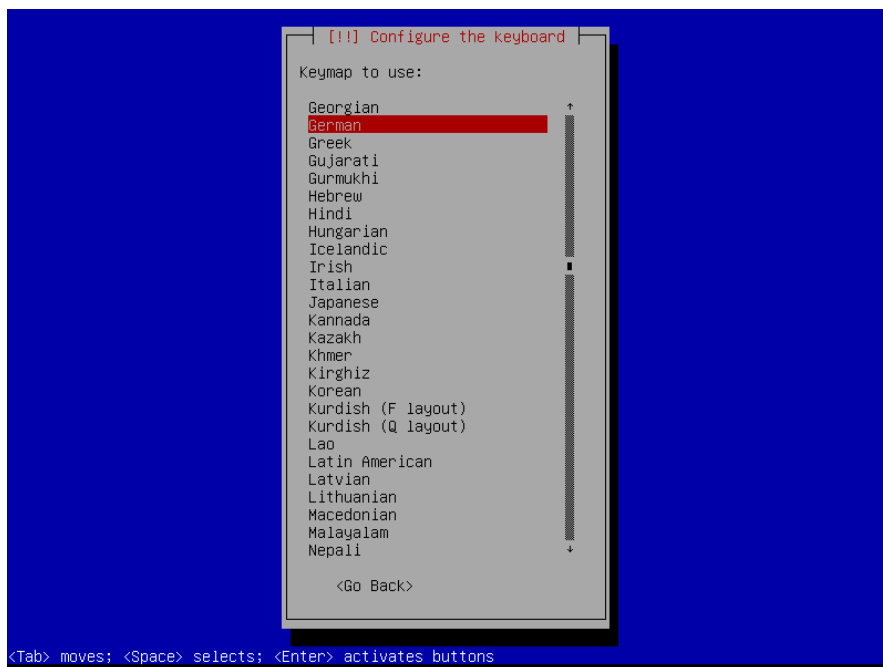


Abb. 9.9 Keyboard-Layout-Auswahl (Debian)

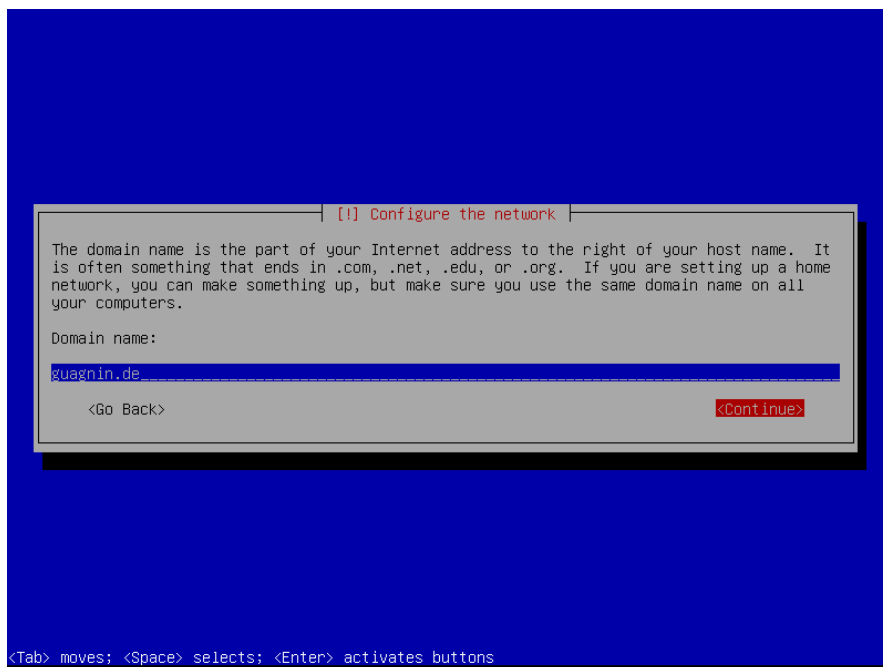


Abb. 9.10 Einrichtung der „Domain“ (Debian)

Dabei wird kurz erläutert, dass der Rechnername zur Identifizierung im Netzwerk dient und dies im Heimnetzwerk beliebig ausgefüllt werden kann. Dasselbe gilt für die *Domain*, jedoch wird angemerkt, dass die Rechner eines Netzwerks derselben Domain zugeordnet werden sollten. Hier ist also eine gewisse Auseinandersetzung mit der Bestimmung des Rechners und der Verortung im Kontext (Heimnetzwerk oder größeres Netzwerk) notwendig, wenngleich die Eingaben für einen einzelnen Computer unwichtig sind. Eine Adressierung von fortgeschrittenen Usern ist hier klar erkennbar.

Auch sind die Informationen für einen User, der lediglich über *Beer-Mat Knowledge* verfügt, nicht so einfach nachvollziehbar. Im Gegensatz dazu wird dies in Ubuntu im Rahmen von User-Kennung und Passwort abgehandelt. So kann dies in einem Schritt erledigt werden, zudem werden diese Felder mit einem Vorschlag vorausgefüllt (vgl. Abb. 9.7). Hier werden also relativ ausführliche Informationen für das Treffen der Entscheidung bereitgestellt, dort (bei Ubuntu) wird anstelle der Informationen, die für Laien komplex sind, eine Vorauswahl getroffen.

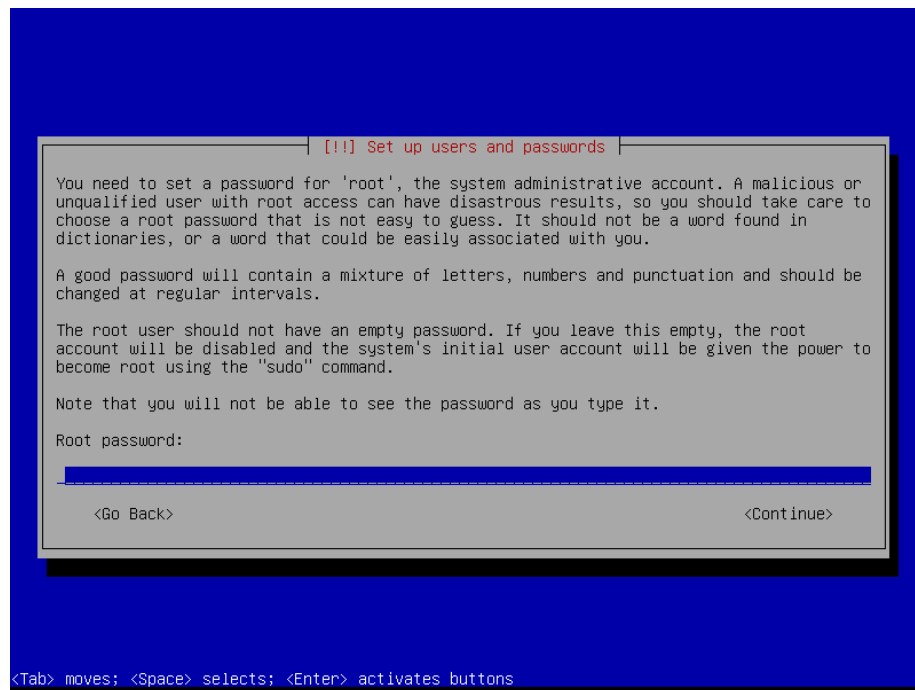


Abb. 9.11 Administrations-Konto *root* oder *sudo* (Debian)

Vor der Einrichtung des User-Kontos steht beim Debian Installer noch die Vergabe eines *root*-Passworts an. Es wird im erläuternden Text darauf hingewiesen, dass dies zur Administration des Computers dient und daher nicht in falsche (böartige oder unqualifizierte) Hände kommen sollte (Abb. 9.11; S. 230). Hier wird also dem User implizit eine gewisse Qualifikation unterstellt.

Hier wird auch die Möglichkeit gegeben, durch ein leeres *root*-Passwort den traditionellen Administrations-Account zu deaktivieren und stattdessen die Administration durch den normalen User-Account zu ermöglichen (über *sudo*, s.o.). Die Substitution des Administrator-Accounts durch *sudo* wurde von Ubuntu für das einfachere Verständnis von Umsteiger*innen von Anfang an angewendet und hat mittlerweile auch Eingang in die Konfigurationsoptionen des Debian-Installers erhalten – *sudo* ist hier aber nicht fest vordefiniert, sondern kann in diesem Schritt vom User selbst gewählt werden. Bemerkte sei an dieser Stelle, dass die Installationsschritte zum Teil mit längeren Informationstexten einhergehen, die dem User nach Möglichkeit den Hintergrund der Konfigurationsmöglichkeiten erklären. Die Bereitstellung der Informationen soll ihm eine informierte Entscheidung über nicht immer triviale Zusammenhänge ermöglichen. Im Vergleich zu Ubuntu ist hier die Einflusskomponente *Information* sehr stark, während bei Ubuntu Informationen stark reduziert sind und so die Standardauswahl motiviert wird – ein *Anreiz*, der Voreinstellung zu folgen, gegenüber einer eigenen Konfiguration mit ungewissem Ausgang („something else“, vgl. Abb. 9.4).

Danach folgen die Festlegung der Zeitzone und anschließend die Konfiguration der Festplatten (Abb. 9.12; S. 232). Hier werden dem User drei geführte Partitionierungsmodi angeboten. Im gezeigten Beispiel erkennt der Installer eine leere Festplatte und schlägt drei alternative Verfahren vor: eine einfache Partitionierung, eine Installation über sogenannte Logical Volumes¹⁴⁰ oder aber die Einrichtung einer Festplattenverschlüsselung.

¹⁴⁰ eine Art Partitionierung, die sich dynamisch verändern lässt, siehe https://de.wikipedia.org/wiki/Logical_Volume_Manager, letzter Aufruf 7.4.2018

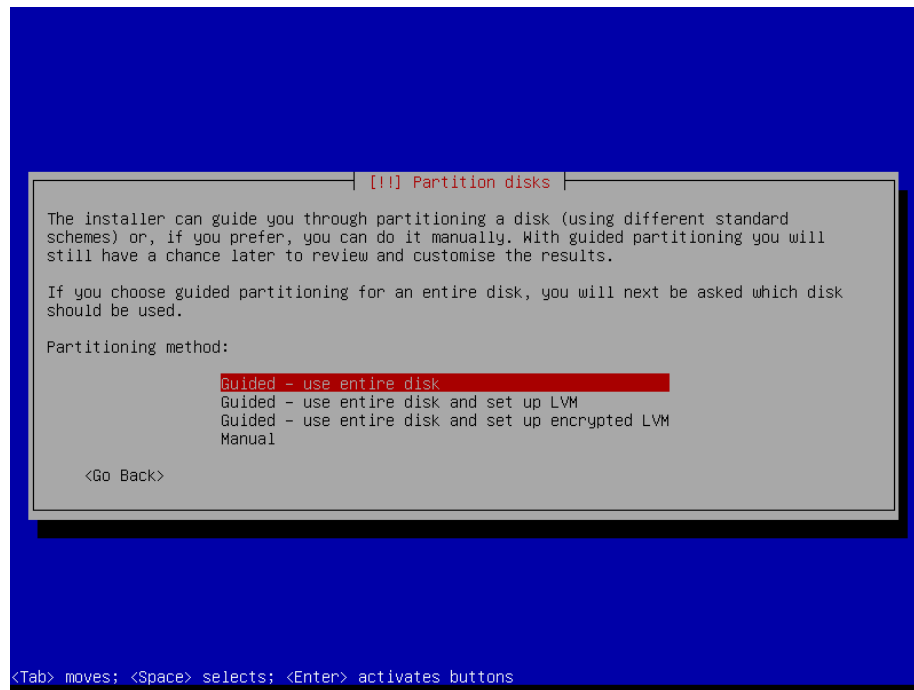


Abb. 9.12 Einrichtung der Festplatte (Debian)

Generell bekommt der User eine andere Art von Informationen zur Verfügung gestellt als im Falle von Ubuntu. Die transportierten Inhalte sind teils genauer und komplexer, manche Zusammenhänge werden implizit als bekannt vorausgesetzt (*Adressierung* und *Voraussetzungen* von Usern). Schließlich zeigt sich hierbei auch ein größerer Konfigurationsspielraum und spiegelt eine geringere Fokussierung des Anwendungszwecks (*Spezifität*): Der Installer ermöglicht sowohl die Installation eines umfangreichen grafischen Desktop-Systems als auch eines minimalen Server-Systems. Schließlich zeigt sich hier die Adressierung eines erfahrenen User-Typs beziehungsweise anspruchsvoller Power-User oder Systemadministrator*innen. Im Laufe der geführten Partitionierung (Abb. 9.12) wird nachgefragt, ob die Festplatte für verschiedene Funktionsbereiche partitioniert werden soll. Hierbei werden übliche Partitionierungen vorgeschlagen (Abb. 9.13; S. 233). Neben den Optionen eines separaten Bereichs für User-Daten und weiterer funktionaler Bereiche wird die erste Option explizit für *Anfänger* beziehungsweise „new users“ empfohlen – alle Daten sollen dabei auf einen Festplattenbereich geschrieben werden.

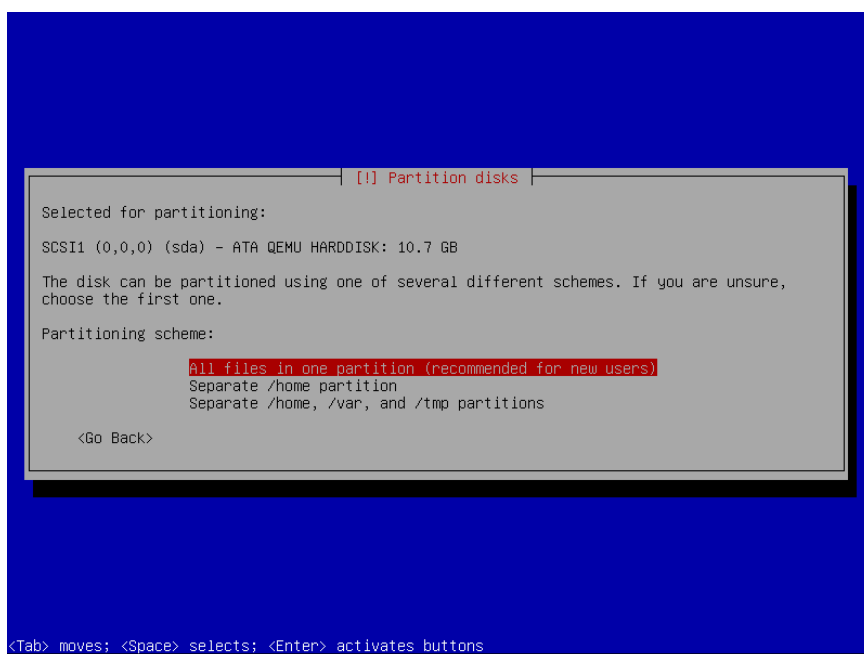


Abb. 9.13 Einteilung der Festplattenbereiche (Debian)

Ein bedeutender Unterschied zum Fall Ubuntu ist außerdem das unterschiedliche Skriptverhalten im Anschluss an die automatische Partitionierung. Hier wird nämlich das Ergebnis der geführten Partition detailliert angezeigt und es können weitere Änderungen vorgenommen werden, bevor das Grundsystem installiert wird (Abb. 9.14; S. 234). Anschließend wird das Grundsystem installiert und somit entsteht schon ein lauffähiges Betriebssystem, das allerdings noch keine grafische Oberfläche besitzt.

Nach der Installation des Grundsystems kann in einem weiteren Schritt definiert werden, welchen Zweck das System erfüllen soll (Abb. 9.15; S. 234). Es werden einige Optionen vorgeschlagen, die jeweils eine Reihe von Optionen enthalten, die mit dem jeweiligen Zweck verbunden sind.¹⁴¹

¹⁴¹ Ohne Internet-Zugriff wird hier nur eine reduzierte Auswahl angezeigt, da die CD nur die Basis-Pakete enthält. Jedoch ist es möglich, weitere CDs einzubinden, die weitere Softwarepakete enthalten – eine Offline-Installation ist also grundsätzlich vorgesehen.

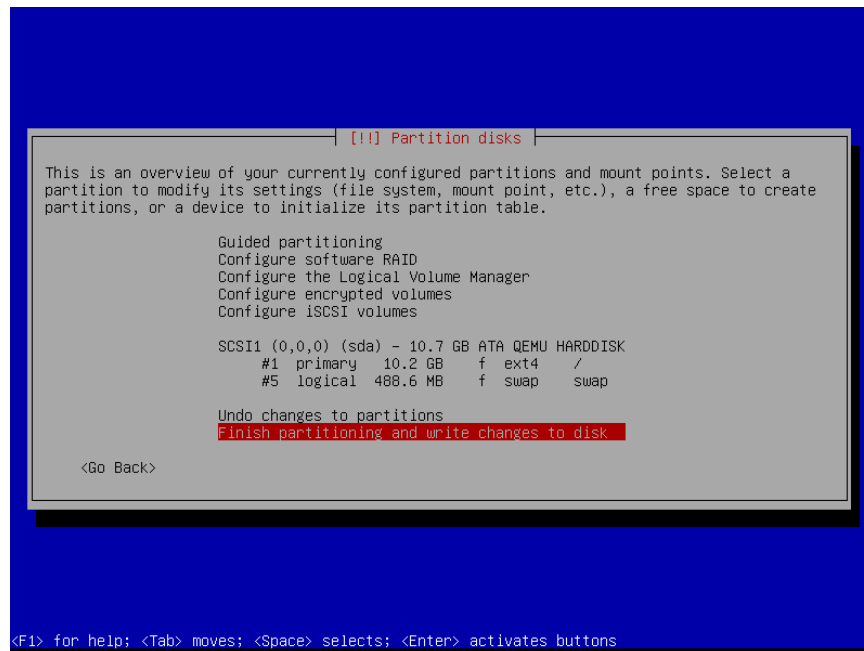


Abb. 9.14

Vorgeschlagene Konfiguration und Möglichkeit der Nachjustierung (Debian)

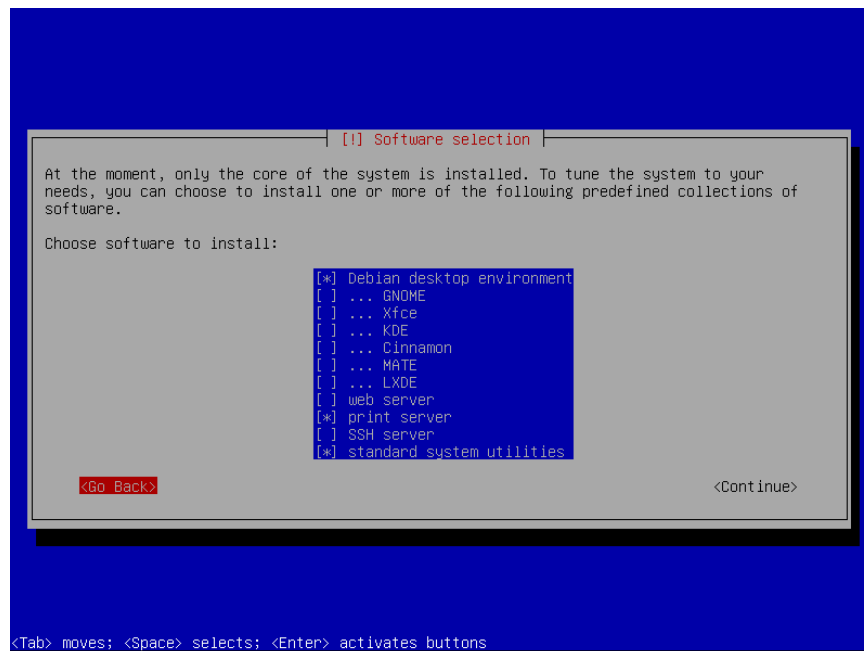


Abb. 9.15 Auswahl der zu installierenden Software

Neben der Auswahlmöglichkeit eines *Desktop Environment*, die die für einen Desktop-Arbeitsplatz übliche Software enthält (Office-Anwendungen, Internet-Browser etc.), kann gewählt werden aus einer Liste verschiedener grafischer Oberflächen (*GNOME*, *Xfce*, *KDE*, ... – bei Linux stehen hier verschiedene zur Auswahl) und diverse Server-Pakete. Die *Spezifität des Zwecks* des resultierenden Systems ist also recht breit angelegt und kann hier über die Auswahl bestimmt werden. Eine grafische Oberfläche ist bemerkenswerterweise nicht vorausgewählt. Wird hier keine Veränderung vorgenommen, wird keine grafische Oberfläche installiert. Für einige Anwendungsfälle reicht ein solches Minimalsystem aus, für einen typischen Desktop-User ist die reine Kommandozeile jedoch wenig hilfreich – es können aber dann gezielt die einzelnen Programmpakete installiert werden, die gewünscht sind. Ein gewisses Vorwissen, für welche grafischen Oberflächen die Abkürzungen *GNOME*, *Xfce*, *KDE*, etc. stehen, ist also implizit gefordert, gesetzt den Fall, man möchte darauf nicht verzichten – ebenfalls eine Wahl, die dem User im Rahmen des Ubuntu-Installers erspart wird.¹⁴²

Schließlich wird noch gefragt auf welche Festplatte der *Bootloader* installiert werden soll – das Programm, das üblicherweise nach Start des Computers das Betriebssystem selbst startet (Abb. 9.16). Hierzu zeigt der Installer im gezeigten Beispiel die Information, dass kein anderes System auf dem Rechner gefunden wurde, und den Hinweis, dass bei der Installation von verschiedenen Systemen auf einem Rechner sichergestellt werden muss, dass alle Systeme gestartet werden können und sich hier nicht gegenseitig behindern.

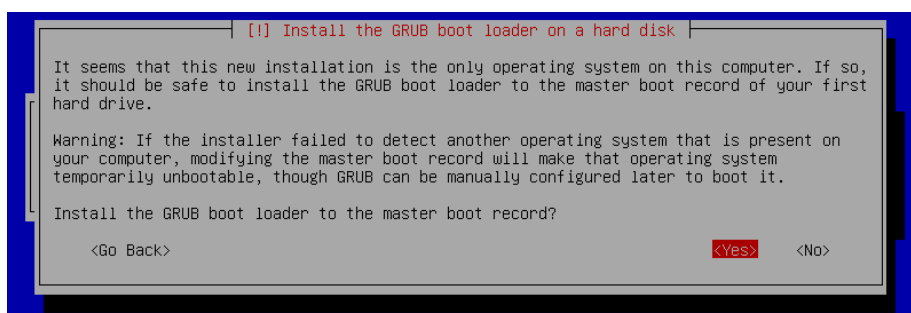


Abb. 9.16 Installation des Boot Loaders

142 Vielmehr gibt es dort spezifische Installationsmedien für jede Oberfläche.

Es wird deutlich, dass hier wiederum gewisse Kenntnisse über installierte Systeme und deren notwendigen Konfigurationen für den Systemstart hilfreich sind. Zumindest ist eine gewisse Gelassenheit gegenüber den möglichen Folgen einer Fehleingabe hilfreich. Hier unterscheidet sich sicherlich ein erfahrener User, der schon verschiedene Fehlinstallationen erlebt (und repariert) hat, von Linux-Einsteiger*innen, die möglicherweise (irrationale) Befürchtungen haben, ihr System irreparabel zu beschädigen. Anschließend ist das frisch installierte System bereit und der Rechner kann mit diesem neu gestartet werden.

Bemerkenswert im Vergleich zur strikten Assistenten-Struktur von Ubuntu ist die Möglichkeit, im Debian Installer auf ein Hauptmenü zu gelangen, von dem aus jeder einzelne Schritt der Konfiguration erreicht werden kann. Auch in der Sequenz der Konfigurationsschritte zeigt sich hier also weniger *Zwang* als mehr Wahlfreiheit.

Die Beschreibungen im Installationsprozess verweisen auf Begriffe, die Vorwissen erfordern (*Adressierung*), und die Konfigurationsoptionen ermöglichen neben der Nutzung eines Desktop-Systems auch verschiedene andere Zwecke (*Spezifität*). Die weiteren Inskriptionen fasse ich im nächsten Abschnitt in einer Tabelle zusammen.

9.2.3 Inskriptionen des Debian-Installers

Die Inskriptionen nach dem beschriebenen Analyseraster finden sich in Tabelle 9.2 (S. 237).

Das Skript impliziert eine *Adressierung* an erfahrenere User wie Power-User oder Administrator*innen, die ein gewisses Verständnis von Computern haben, wie es beispielsweise in Computerzeitschriften vermittelt wird (*Popular Understanding*).

Der *Zweck des Skriptes* ist recht unspezifisch, das Zielsystem (der Computer) kann für unterschiedliche Einsatzzwecke – vom Arbeitsplatzrechner bis zum Web-Server – konfiguriert werden.

Art des Einflusses: Informationen werden bereitgestellt, um eine informierte Entscheidung zu ermöglichen. Der Installationsassistent ist klar strukturiert, über ein Hauptmenü kann die vorgesehene Sequenz aber verlassen werden, um gezielt einzelne Schritte anzuwählen. Der *Point of no return* des Schreibens auf die Festplatte geschieht sehr spät, bis dahin kann die Installation also noch problemlos abgebrochen werden.

Die *Verteilung der Kontrolle* wechselt zwischen User und Skript; der User hat also eine mittlere Flexibilität.

Der Installationsverlauf und das Ergebnis hängen ab von der *Gegebenheit* einer Internet-Verbindung oder weiteren Installations-CDs, da auf dem Medium nicht die Dateien für ein vollständiges *grafisches* Desktop-System enthalten sind.

Während der Skript-Phasen sind die konkreten Vorgänge und Befehle, die das Skript ausführt, nicht *sichtbar*, die Beschreibungstexte der Installations-skripte stellen aber Informationen darüber bereit.

Tab. 9.2: Zusammenfassung der Inskriptionen des Debian-Installers

Dimension	Fall Debian
Adressierung/Voraussetzungen von Usern	Power-User, <i>Popular Understanding</i>
Spezifität des Zwecks	mittel: verschiedene Systemarten installierbar (von Server bis Desktop)
Art des Einflusses	<i>Informationen</i> dienen der Auswahl vorgegebener Optionen (<i>Ausstattung</i>); später <i>Point of no return</i>
Verteilung der Kontrolle	wechselnd zwischen User und Skript; verhältnismäßig viele Konfigurationsmöglichkeiten
Homogenität der Kontrollverteilung	über den gesamten Verlauf wechseln sich Eingabe von Konfigurationsdaten und das Warten auf den nächsten Schritt ab
materielle Gegebenheiten des Kontextes	Internet benötigt, um ein vollständiges Desktop-System zu erhalten
Sichtbarkeit des Skripts	Durch die Schrittweise Skriptführung ist jederzeit der aktuelle Installationsschritt (abstrahiert) sichtbar, die konkreten Befehle sind dem User aber verborgen.

9.3 Arch Linux: Volle Kontrolle – do it yourself

Hier offenbart sich der *Installer* mit einer Befehlszeile und Hinweisen auf die Dokumentation, in der die notwendigen Befehle zu finden sind. Diese Befehle darf der User selbst eintippen, er hat somit die volle Kontrolle und wird faktisch selbst zum Installations-Skript.

Arch folgt dem Leitsatz „Keep it simple, stupid“ (KISS). Dieses Verständnis von technischer „Einfachheit“ bringt es offenbar mit sich, dass User hier mehr über die Funktionsweise lernen als bei Verfolgung eines Usability-Ansatzes, der außerdem häufig technisch komplexer in der Implementation ist. Dies führt schließlich zu mehr „Kontrolle“ der User über das System:

I use it a lot, is as a learning tool, I mean, there is a lot of overhead, you have to have free time to run it, because you constantly update things and change configs, and it's just lot of your time. That being said, whether you use it as learning tool, or you use it because you *really* want control over your system like complete control, and so it's kind of a range of less control, more control. (4,128)

9.3.1 Getting and Installing Arch

Auf der Arch-Linux-Website findet sich ein Link auf den „Installation Guide“¹⁴³, was das eigentliche Installationskript darstellt – Skript im Sinne von Beschreibung. Hier wird beschrieben, in welcher Reihenfolge welche Dinge erledigt werden sollten und welche Befehle dabei behilflich sind. Diese werden dabei häufig nicht im Detail ausgeführt. Vielmehr müssen User die Details aus den entsprechenden Manuals heraus extrahieren, um eine erfolgreiche Installation von Arch zu erhalten. Unter der Rubrik „Pre-Installation“ ist hierbei die Seite „Getting and Installing Arch“¹⁴⁴ verlinkt, über die man das Installationsmedium beziehen kann.

Dabei wird wie bei Debian empfohlen, das Image über *BitTorrent* herunter zu laden; aber es gibt auch die Möglichkeit, das Image über http zu laden. Wie bei Debian wird hier außerdem darauf hingewiesen, dass überprüft werden sollte, ob das Medium, das heruntergeladen wurde, auch wirklich das von Arch Linux angebotene ist und nicht etwa eine kompromittierte Datei, in die beispielsweise Hintertüren eingebaut sein könnten.

Die Informationen auf der Download-Seite sind hier im Vergleich zu Debian spärlich und reduziert. Einzig fällt auf, dass es ein spezielles Netboot-Medium gibt, das technisch ausgefeilt mit einem minimalen Speicher auskommt (1 MB!) und über „iPXE“ die vollständige Installation aus dem Internet zieht. Dies impliziert, dass ein User selbst weiß, wie er welches Medium zu laden hat – und möglicherweise auch Interesse, eine technisch raffinierte Installationsvariante zu nutzen (*Adressierung*).

143 https://wiki.archlinux.org/index.php/Installation_guide

144 https://wiki.archlinux.org/index.php/Category:Getting_and_installing_Arch

9.3.2 Being the Install-Script

Der Startbildschirm des Arch Installer hat zunächst Ähnlichkeit mit dem Debian Installer (Abb. 9.17). Nach dem Start des Installationsmediums findet sich der User allerdings auf einer schlichten Kommandozeile wieder, auf der er nun die nötigen Schritte zur Installation selbst durchführen kann (Abb. 9.18; S. 240). Der User hat so über den kompletten Installationsprozess die volle Kontrolle, einzig laufen die gestarteten Programme nach ihrem Aufruf weiter – etwa der Kopierbefehl – bis zur nächsten Eingabe (*homogene Kontrollverteilung: beim User*).

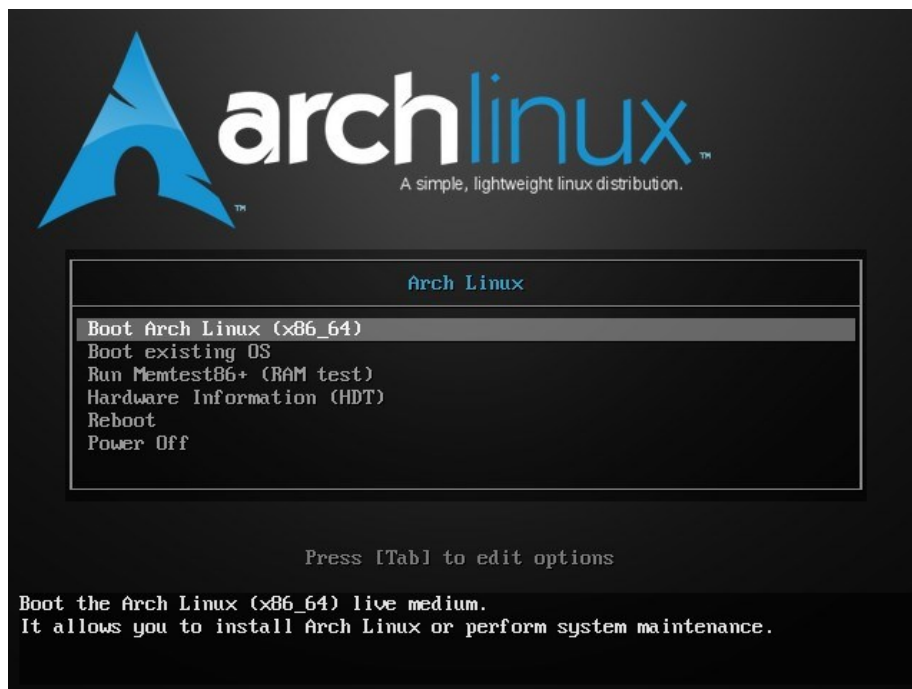


Abb. 9.17 Arch-Linux-Startbildschirm

In einer Datei *install.txt* ist das Vorgehen beschrieben, das in ähnlicher Weise auch im Arch-Linux-Wiki zu finden ist. Das Inhaltsverzeichnis der Datei gliedert sich dabei in die Bereiche *Pre-Installation*, *Installation*, *Systemkonfiguration*, *Reboot* und *Post-Installation* (Abb. 9.19; S. 240). Von den Installationsschritten werde ich im Folgenden exemplarisch einige kurz beschreiben, um den Vergleich zu den oben beschriebenen Fällen zu ziehen.


```

Arch Linux 4.10.6-1-ARCH (tty1)

archiso login: root (automatic login)
root@archiso ~ # loadkeys de-latin1
root@archiso ~ # fdisk -l
Disk /dev/sda: 10 GiB, 10737418240 bytes, 20971520 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: dos
Disk identifier: 0x04a1ab4b

Device      Boot Start      End  Sectors  Size Id Type
/dev/sda1                2048 16779263 16777216   8G 83 Linux

Disk /dev/loop0: 367.1 MiB, 384892928 bytes, 751744 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
root@archiso ~ #

```

Abb. 9.18 Erste Schritte in Arch, die Festplatte enthält hier schon eine Partition

- * 1 Pre-installation
 - + 1.1 Set the keyboard layout
 - + 1.2 Verify the boot mode
 - + 1.3 Connect to the Internet
 - + 1.4 Update the system clock
 - + 1.5 Partition the disks
 - + 1.6 Format the partitions
 - + 1.7 Mount the file systems
- * 2 Installation
 - + 2.1 Select the mirrors
 - + 2.2 Install the base packages
- * 3 Configure the system
 - + 3.1 Fstab
 - + 3.2 Chroot
 - + 3.3 Time zone
 - + 3.4 Locale
 - + 3.5 Hostname
 - + 3.6 Network configuration
 - + 3.7 Initramfs
 - + 3.8 Root password
 - + 3.9 Boot loader
- * 4 Reboot
- * 5 Post-installation

Abb. 9.19 Notwendige Schritte zur Installation laut mitgelieferter Datei *install.txt*

Als ersten Schritt bietet es sich an, die Tastatureinstellungen anzupassen; da im Folgenden einige Angaben per Text eingegeben werden müssen, ist dies ein wichtiger Schritt. Um herauszufinden, wie die möglichen Optionen heißen, empfiehlt die Installationsanweisung den Befehl:

```
ls /usr/share/kbd/keymaps/**/*.map.gz
```

Anschließend kann man mit dem Befehl `loadkeys de-latin1` die deutsche Tastaturbelegung laden. Für die Eingabe des ersten Befehls muss man aber erstmal herausfinden, wie man die Zeichen `/.y` auf einer deutschen Tastatur findet, die zunächst als amerikanische Tastatur interpretiert wird. Gibt man nämlich die Zeichen an, die auf der deutschen Tastatur angegeben sind, erscheint Folgendes:

```
ls &usr&share&kbd&keymaps&}}&}.map.gy
```

Um also auf einer deutschen Tastatur den gewünschten Befehl zu erhalten, solange noch die englische Tastatur geladen ist, muss man auf dem deutschen Layout Folgendes eingeben, um den korrekten Befehl zu erhalten:

```
ls -usr-share-kbd-keymaps-( (.map.gy
```

```
root@archiso ~ # ls
/usr/share/kbd/keymaps/i386/qwertz/*.map.gz
/usr/share/kbd/keymaps/i386/qwertz/croat.map.gz
/usr/share/kbd/keymaps/i386/qwertz/cz-qwertz.map.gz
/usr/share/kbd/keymaps/i386/qwertz/cz-us-qwertz.map.gz
/usr/share/kbd/keymaps/i386/qwertz/de_alt_UTF-8.map.gz
/usr/share/kbd/keymaps/i386/qwertz/de_CH-latin1.map.gz
/usr/share/kbd/keymaps/i386/qwertz/de-latin1.map.gz
/usr/share/kbd/keymaps/i386/qwertz/de-latin1-nodeadkeys.map.gz
/usr/share/kbd/keymaps/i386/qwertz/de.map.gz
/usr/share/kbd/keymaps/i386/qwertz/de-mobii.map.gz
/usr/share/kbd/keymaps/i386/qwertz/fr_CH-latin1.map.gz
/usr/share/kbd/keymaps/i386/qwertz/fr_CH.map.gz
/usr/share/kbd/keymaps/i386/qwertz/hu.map.gz
/usr/share/kbd/keymaps/i386/qwertz/sg-latin1-lk450.map.gz
/usr/share/kbd/keymaps/i386/qwertz/sg-latin1.map.gz
/usr/share/kbd/keymaps/i386/qwertz/sg.map.gz
/usr/share/kbd/keymaps/i386/qwertz/sk-prog-qwertz.map.gz
/usr/share/kbd/keymaps/i386/qwertz/sk-qwertz.map.gz
/usr/share/kbd/keymaps/i386/qwertz/slovene.map.gz
root@archiso ~ # loadkeys de-latin1
```

Abb. 9.20 Auswahl der qwertz-Keymaps-Dateien für i386-Prozessorarchitekturen

Anschließend muss man sich aus der Liste noch die passende Keymap-Datei heraussuchen. Aufgrund der langen Liste der möglichen Eingaben ist

es hier hilfreich zu wissen, dass man über die Tastenkombination <Shift>+<Bild rauf> den Bildschirmtext nach oben scrollen kann. Alleine für die übliche Rechnerarchitektur i386 (bzw. verwandte Architekturen) und für das qwertz-Layout gibt es 18 verschiedene Layouts. Aus Platzgründen enthält Abbildung 9.20 (S. 241) nur die entsprechende Auswahl i386/qwertz/*. Hier zeigt sich also die Notwendigkeit einer Menge von implizitem Wissen, das sich nicht ohne Weiteres aus den genannten Dokumentationen erschließt.

```
fdisk /dev/sda
Command (m for help): n
Partition type:
p primary
e extended
Select (default p): p
Partition number (1-4, default 1): 1
First sector (2048-..., default 2048):
Last sector, +sectors or +size{K,M,G} (..): +8G
Partition 1 of type Linux and of size 8 GiB is set
Command (m for help): a
Selected Partition 1
The bootable flag on partition 1 is enabled now
Command (m for help): n
[ ... wie zuvor bei Partition 1 ]
Command (m for help): t
Partition number (1-4): 2
Hex code (type L to list codes): 82
Command (m for help): p
Device      Boot Start      End          Sectors    Size Id System
/dev/sda1  *        2048        167772216  16777216  8G  83 Linux
/dev/sda2             16772964   20971519    4192256   2G  82 Linux
                                swap
Command (m for help): w
```

Abb. 9.21 Partitionierung der Festplatte mittels *fdisk* (Arch)

Als nächstes steht die Einrichtung der Festplatte an. Hierzu muss man wissen, ob das System mit der neueren Firmware EFI oder mit dem traditionellen BIOS¹⁴⁵ läuft (Schritt 1.2 in Abb. 9.19). Abhängig davon kommen auch

145 Da die genaue Bedeutung der hier genannten Abkürzungen nicht für das Verständnis des Installers notwendig ist, verweise ich auf https://de.wikipedia.org/wiki/Unified_Extensible_Firmware_Interface, letzter Aufruf 7.4.2018.

unterschiedliche Werkzeuge für die Partitionierung der Festplatte zum Einsatz: *gdisk* (EFI) oder *fdisk* (BIOS). Alternativ gibt es auch das Programm *gparted* für die Partitionierung der Festplatte. Im Beispiel verwenden wir *fdisk* (Abb. 9.21; S. 242).

Die Einrichtung der Festplatte geschieht also nicht per Auswahl im Menü und mit vorgeschlagener Aufteilung, sondern manuell über ein Partitionierungs-Programm, das man selbst auswählen darf. In das verwendete Programm ist dabei schon eine Hilfefunktion eingeschrieben, die dem User ermöglicht, über Tastaturkürzel bestimmte Funktionen auszuführen.¹⁴⁶

Mit *n* wird eine neue Partition generiert; dafür muss man angeben, ob es eine *primäre* oder *extended* Partition sein soll. Hierzu sollte man also wissen, dass es aus historischen Gründen bis zur Erneuerung der Firmware durch *EFI* (Extended Firmware Interface) nur vier Partitionen gab, die bis heute *primär* heißen. Aufgrund der Notwendigkeit von mehreren Partitionen kann eine dieser vier Partitionen eine sogenannte *erweiterte* Partition sein, die wiederum weitere (logische) Partitionen beherbergt. Hilfreich für die Spezifizierung der Größe ist das Wissen, dass Festplatten in Sektoren aufgeteilt sind, wobei jeder Sektor eine definierte Speichergröße hat. Das Programm *fdisk* gibt hier schon den ersten beschreibbaren Sektor an, der einfach nur bestätigt werden muss, und zur Spezifizierung der Speichergröße ist es nicht notwendig, den Endsektor der Festplatte anzugeben, sondern man darf auch die gewünschte Größe in Kilo-, Mega- oder Gigabyte angeben (die Einheiten unterscheiden sich je um den Faktor 1000). Das Programm enthält also durchaus einige vereinfachende Voreinstellungen, wenngleich die Partitionierung dem User doch einiges an Wissen abverlangt.

Mit dem Kürzel *a* wird eine Partition als diejenige Partition markiert, von der das System gestartet wird. Mit *t* wird der Typ definiert, also welche Funktion oder welches Dateisystem die Partition erhält. *82* steht hier für eine Partition zur Auslagerung von Speicher (wenn der Zwischenspeicher (RAM) voll ist, wird diese Partition als erweiterter Zwischenspeicher verwendet).

Nach der Einteilung der Festplatte muss noch das Dateisystem mit dem Programm *mkfs* formatiert und die formatierte Partition schließlich als Zielpartition *eingebunden* werden, damit dorthin die Dateien des neuen Systems kopiert werden können.

Es wird deutlich, dass es kaum möglich ist, in wenigen Worten unter Vermeidung von Fachtermini die Installation zu beschreiben. In den oben

146 Die Tastenkürzel werden über die Taste *m* angezeigt.

beschriebenen Fällen geschieht die Installation durch die Auswahl verschiedener Menüpunkte des Skriptes. Dort werden die notwendigen Befehle nach der getätigten Auswahl vom Skript automatisch ausgeführt, hier bei Arch hingegen werden sie per Hand vom User eingegeben. Dadurch lernt der User im Zuge der Installation sein System (zwangsläufig) kennen. Programme, die er nicht kennt, muss er sich aneignen und mit Konzepten, die ihm unbekannt sind (wie etwa Partitionierung, *fdisk*), muss er sich vertraut machen, um die Installation erfolgreich zu bewältigen. Die nötigen Informationen finden sich in den systemeigenen Hilfedateien (*Man-Pages*) und der *install.txt* oder auf der entsprechenden Arch-Wiki-Seite, wo jeweils Links die Vertiefung der einzelnen Punkte ermöglichen. Der User hat dadurch jederzeit die Kontrolle über den nächsten Befehl, und die Befehle mitsamt ihren einzelnen konfigurierenden Parametern sind im Skript voll *sichtbar*.

Im Laufe des Installationsprozesses konfiguriert der User in Schritt drei der Installationsanleitung (Abb. 9.19) das neue System zunächst ohne Neustart mittels des Werkzeuges *chroot*. Dort werden für das neue System nochmals die Tastaturbelegung, Zeitzone und Sprache sowie das Netzwerk konfiguriert. Die Konfiguration des Rechnernamens gestaltet sich im Vergleich zu den oben beschriebenen Aufgaben ganz „simpel“ mittels der Erstellung einer Datei (*/etc/hostname*), die den Rechnernamen enthält. In Abbildung 9.22 findet sich der entsprechende Auszug aus den Installationshinweisen.

```
Hostname
Create the hostname(5) file:
/etc/hostname myhostname

Consider adding a matching entry to hosts(5):
/etc/hosts
127.0.0.1      localhost.localdomain  localhost
::1          localhost.localdomain  localhost
127.0.1.1    myhostname.localdomain myhostname

See also Network configuration#Set the hostname.
```

Abb. 9.22

Auszug aus der Arch-Installationsanweisung zur Konfiguration des Rechnernamens

Hier zeigt sich also, dass allein für die Installation ein hohes Maß an Wissen erforderlich ist oder zumindest die Bereitschaft, sich das notwendige

Wissen anzueignen. Die Installation kann in dieser Hinsicht als eine Art Initiation betrachtet werden, in der die User ihre Fähigkeiten zur Installation unter Beweis stellen. Dafür spricht auch, dass ehemals vorhandene Skripte zur Vereinfachung sich nicht durchgesetzt haben. Abgeleitete Distributionen, die auf Arch basieren, aber einen einfachen Installer anbieten, sind nicht unbedingt erwünscht in der Arch Community.¹⁴⁷

In einer Diskussion auf der Mailingliste im Juli 2016 stellt ein User fest, dass an die Stelle der automatisierten Skripte eine Wiki-Seite getreten ist, die „arch-install-scripts“ lautet (die im Übrigen heute weiterleitet auf den oben referenzierten *Installation Guide*) und die Anweisungen nicht automatisiert an den Rechner, sondern textbasiert an den User weitergibt. Er bekommt eine klare Antwort, dass die Auseinandersetzung mit der Installation notwendig ist, um Arch User zu werden:¹⁴⁸ Die Methoden, die zur Installation notwendig sind, helfen dem User später auch, Probleme zu finden und zu beheben, so das Credo der Community.

- > I've read the wiki page for arch-install-scripts. It is not very
- > "user-friendly". Any hope to see this "embedded" in a program,
- > like good old arch/setup ?

Personally I think we would be better off with just these scripts + lots of documentation. If you are interested in an installer, then I guess the best solution is to get involved with the AIF and get it into shape again.

- > I'm not afraid of these scripts, but it could drive away some users
- > which are not ready for a so "simplified" installation tool.

I think this is a good thing (once we have proper documentation). It forces users to understand how their system is set up, and how to fix it in case of problems. Unlike with other distributions (e.g. Ubuntu), I think this is a necessity when using Arch, and making this clear from the beginning should hopefully save both us and users from lots of grief.

Das hier referenzierte AIF (Arch Installer Framework) wird allerdings nicht weiter gepflegt. Vor dem Hintergrund der Arch-Philosophie und der damit verbundenen Einstellung der User ist es nachvollziehbar, dass jemand, der sich einmal eingearbeitet hat in die Funktionsweise der Distribution, weder

147 Vergleiche beispielsweise die Diskussion ab Minute 55 zum Vortrag „Arch aus Nutzersicht“ auf den Chemnitzer Linux-Tagen 2016 über das „Canonicalisierte Arch“ *Manjaro*, siehe <https://chemnitzer.linux-tage.de/2016/de/programm/beitrag/345>, letzter Aufruf 18.4.2017.

148 <https://lists.archlinux.org/pipermail/arch-general/2012-July/028191.html>, letzter Aufruf 18.4.2017

Motivation noch Sinn für die nicht triviale Pflege eines derartigen Installations-Werkzeuges hat und seine Energie lieber in andere Dinge steckt.¹⁴⁹

Ebenso generisch wie das Installationsmedium (Kommandozeile) ist schließlich das resultierende System. Zunächst entsteht ein minimales Betriebssystem, das nur die für die Funktion essenziellen Bestandteile enthält, die *Spezifität des Nutzungszwecks* ist also minimal. Vielmehr wird im Installationskript (der Textdatei) eine Phase der Post-Installation vorgesehen, in der die spezifischen Programme wie die grafische Oberfläche, oder Peripheriegeräte wie Sound und Touchpad eingerichtet werden (Abb. 9.23).

```
Post-installation

See General recommendations for system management directions and post-installation tutorials (like setting up a graphical user interface, sound or a touchpad).

For a list of applications that may be of interest, see List of applications.
```

Abb. 9.23 Erst nach der eigentlichen Installation werden die grafische Benutzeroberfläche sowie die Anwendungssoftware installiert.

9.3.3 Inskription des Arch-Installers

Die Inskriptionen nach dem beschriebenen Analyseraster finden sich in Tabelle 9.3 (S. 247).

Die *Adressierung* des Skripts fokussiert auf technisch versierte und interessierte User, die bereit sind, im Zweifelsfall die Dokumentation zu lesen und daher zumindest *Primary Source Knowledge* besitzen müssen.

Die *Spezifität des Zwecks* ist sehr gering. Ziel des Installationsprozesses ist zunächst ein minimales Linux-System, das für jegliche Zwecke mit weiterer Software ausgestattet und erweitert werden kann.

Art des Einflusses: Der User ist an keinen Skriptablauf gebunden, lediglich die technischen Bedingungen für eine erfolgreiche Installation geben einen Rahmen vor. Wer möchte, findet eine *Bereitstellung von Informationen* als Vorschlag für einen erprobten Handlungsablauf für eine erfolgreiche Installation.

149 Vergleiche dazu die Diskussion über das AIF auf reddit von 2014: https://www.reddit.com/r/archlinux/comments/2mfupb/still_remember_arch_linux_installation_framework/?st=jinkjvnf&sh=9b624077#bottom-comments, letzter Aufruf 18.4.2017.

Die *Verteilung der Kontrolle* liegt homogen nahezu ausschließlich beim User, einzig während des Ablaufs einer Befehlseingabe arbeitet der Computer und der User ist passiv. Die Flexibilität der Nutzung ist damit sehr hoch.

Zu den *materiellen Gegebenheiten* zählt die Notwendigkeit einer Internetverbindung für den Bezug der Installationsdateien. Sie hilft auch bei der Suche nach Informationen.

Das Installationskript ist insofern sehr *sichtbar*, als jeder Befehl mitsamt allen Parametern manuell vom User eingegeben wird.

Tab. 9.3: Zusammenfassung der Inskriptionen des Arch-Installers

Dimension	Fall Arch
Adressierung/Voraussetzungen von Usern	technisch versierte und interessierte User, <i>Primary Source Knowledge</i>
Spezifität des Zwecks	minimales (textbasiertes) Live-System, das eine Installation <i>ermöglicht</i> , aber auch für anderes genutzt werden kann
Art des Einflusses	gering; selbst die notwendigen Informationen muss sich der User selbst beschaffen.
Verteilung der Kontrolle	beim User, einzig während der angestoßenen Befehle läuft das aufgerufene Programm
Homogenität der Kontrollverteilung	Kontrolle liegt homogen auf der Seite des Users
materielle Gegebenheiten des Kontextes	Internet-Verbindung notwendig, um die notwendigen Installationsdateien zu beziehen
Sichtbarkeit des Skripts	volle Sichtbarkeit aller Befehle und Parameter durch manuelle Eingabe

9.4 Zusammenfassende Gegenüberstellung der Skripte

Der Vergleich der Installationskripte (Tab. 9.4; S. 248) zeigt die normativen Einschreibungen des Installationskriptes und wie die User durch seine Beschaffenheit informiert werden.

Tab. 9.4: Zusammenfassung der Inskriptionen der Installer

Dimension	Fall Ubuntu	Fall Debian	Fall Arch
Adressierung	Ein- und Umsteiger	Power-User	Experten
Voraussetzung von Usern	<i>Beer-Mat-Knowledge</i>	<i>Popular Understanding</i>	<i>Primary Source Knowledge</i>
Spezifität des Zwecks	hoch: Standard-Desktop-System	mittel: von Server bis Desktop	gering: minimales System
Einflussarten:			
– Zwang	hoch (früher <i>Point of no return</i>)	mittel (Pfad kann verlassen werden)	gering
– Anreiz	hoch	gering	gering
– Ausstattung	hoch (Vorkonfiguration)	hoch (verschiedene Vorkonfigurationen)	gering (Kommandozeile)
– Information	mittel (kurze Infos)	hoch (informierte Entscheidung)	gering (RTFM)
Verteilung der Kontrolle	stark beim Skript	abwechselnd Skript/User	stark beim User
Homogenität der Kontrollverteilung	homogen	wechselnd	homogen
materielle Gegebenheiten des Kontextes	unabhängig von einer Internet-Verbindung.	Internet benötigt für ein vollständiges Desktop-System; Kontext: variables Installationsmedium (von Server bis Desktop)	Internet-Verbindung notwendig, um die notwendigen Installationsdateien zu beziehen
Sichtbarkeit des Skripts	Parallelität von Konfiguration: Kopiervorgang/Installation läuft teils im Hintergrund	Schrittweise Skriptführung, aktueller Installationsschritt sichtbar	volle Sichtbarkeit aller Befehle und Parameter durch manuelles Eingeben

Das Nutzerbild der Communities spiegelt sich deutlich wider in der *Adressierung der Nutzer* sowie der *Spezifität des Zwecks* des Installationsmediums beziehungsweise in den eingeschriebenen spezifischen Nutzungsmöglichkeiten. Die Installations-CD von Ubuntu ist mit nur rudimentärem

Wissen (*Beer-Mat Knowledge*) über Computer nutzbar. In der Debian Installation werden darüber hinaus einige grundlegende Kenntnisse impliziert. Bei Arch hingegen ist eine erfolgreiche Installation nur erfahrenen Usern möglich – zumindest ist dafür eine Auseinandersetzung mit der einschlägigen Dokumentation notwendig (*Primary Source Knowledge*).

Bemerkenswerterweise unterscheidet sich auch die *Spezifität des Zwecks* des Installationsmediums. Zwar sind alle Installationsprogramme dafür da, den Zweck einer Installation des Betriebssystems zu erfüllen, jedoch variieren die Einsatzzwecke und Verwendungsmöglichkeiten des Resultats entsprechend der adressierten Nutzergruppe. Während bei Ubuntu das installierte System als typischer Arbeitsplatzrechner (Desktop-PC) sehr spezifisch ist, kann bei der Debian-Installation zwischen System-Komponenten für verschiedene Einsatzzwecke gewählt werden: Desktop-System, Web- oder Print-Server beispielsweise. Arch schließlich generiert zunächst lediglich ein minimales Kernsystem mit Basisfunktionalitäten. Von dem ausgehend können einzelne gewünschte Programme installiert werden, jenseits des Kernsystems ist aber nichts vorinstalliert.

Die *Art des Einflusses* variiert zwischen den Programmen, aber auch innerhalb der jeweiligen Skripte. Bei Ubuntu liegt in der Bereitstellung eines attraktiven Live-Systems ein *Anreiz*, den User in seiner Entscheidung, das System zu installieren, zu bestärken. Während der Installation kommt der obligatorische Schritt, die Festplatte mit dem System zu bespielen, sehr früh. Die übrige Konfiguration erfolgt parallel zum Installationsprozess, ein Zurück ist an dieser Stelle nicht mehr angedacht. Die Oberfläche der Software ist grafisch ansprechend gestaltet (*Anreiz*) und die Schrittfolge des Installationsassistenten sehr restriktiv (*Zwang*). Bei Debian werden Informationen angeboten, die eine informierte Entscheidung über die Konfiguration ermöglichen sollen. Innerhalb des (grafisch sehr schlicht gestalteten) Installationsassistenten kann jederzeit ein Haupt-Menü aufgerufen werden, von dem aus einzelne Schritte der Installation gezielt gewählt werden können. Hier erfolgt also keine erzwungene Abfolge der einzelnen Skript-Schritte. Vielmehr hat der User mehr Möglichkeiten und wird stärker über Information geleitet als über den *Anreiz* einer ansprechenden Oberfläche. Die Installationsassistenten von Debian und Ubuntu stellen eine starke *Ausstattung* dar, die Usern bestimmte Installationsoptionen an die Hand gibt, während bei Arch lediglich die Kommandozeile zur Verfügung steht. Bei Arch gibt es also kein automatisches Skript, das bestimmte Schritte vorgibt oder überhaupt Befehle im Hintergrund ausführt, und es gibt auch keine grafische Oberfläche. Schließ-

lich muss man sogar die Informationen, die jemandem einen Vorschlag über den Installationsablauf unterbreiten, erst selbst finden.

Folglich ist die *Verteilung der Kontrolle* bei Arch homogen auf der Seite des Users, der volle „Befehlsgewalt“ hat und lediglich nach Befehlseingabe jeweils auf die Beendigung des Programms warten muss. Bei Debian sind die konkreten Befehle nicht im Detail modifizierbar, der User büßt hier an Kontrolle ein, kann aber von Schritt zu Schritt den Verlauf des Prozesses beeinflussen. Bei Ubuntu gibt der User mit der Bestätigung des Schreibzugriffs auf die Festplatte die Kontrolle weitestgehend ab.

Ein Unterschied in der Rolle der *materiellen Gegebenheiten des Kontextes* findet sich in der Abhängigkeit einer Internet-Verbindung. Nur bei Ubuntu sind auf dem Installationsmedium die Softwarepakete für ein grafisches Desktop-System vorhanden, die anderen laden dies nur bei Bedarf aus dem Internet oder von weiteren CDs.

Die *Sichtbarkeit* ist schließlich vor allem bei Arch gegeben, wo jeder Befehl manuell eingegeben wird. Bei Debian geschieht dies im Verborgenen, aber der Verlauf der Installation enthält mehr Informationen über die Konfigurationsschritte und ist durch den linearen Ablauf leichter nachvollziehbar als bei Ubuntu, wo parallel, während die Installation schon läuft, noch Eingaben für die weitere Konfiguration abgefragt werden.

Besonders stark unterscheiden sich die Skripte im Maß an Kontrolle, die dem User zugetraut oder auch zugemutet wird.

Besonders Interessant ist in diesem Zusammenhang die Einflussdimension *Art des Einflusses*: Beim Arch-Skript gibt es im Grunde keine Einflussnahme auf den User, die ihn dazu bringt, das System zu installieren. Er hat die volle Kontrolle über den Installationsprozess und kann auf dem generischen Installationsmedium auch irgendetwas anderes tun. Dadurch ist der User wiederum gezwungen, sich mit der Funktionsweise des Systems auseinanderzusetzen und sich gegebenenfalls die Dokumentation anzueignen – allerdings nur, wenn er das System auch *wirklich* installieren will.

Die völlige Handlungsfreiheit der Experten steht hier im Gegensatz zur Ermöglichung einer spezifischen Installation für Laien durch die Begrenzung der Freiheitsgrade im Skript. Ebenfalls liegt hierin eine Exklusion oder Inklusion von Laien begründet, die entweder als inkompetent und nicht beitragsfähig erachtet werden, oder aber als Potenzial gesehen werden, die auf ihre eigene Weise mit ihren Fähigkeiten beitragen und dabei helfen, ein laienfreundliches Produkt mitzugestalten.

Der Vergleich zeigt auch, dass diese Grenzziehung Gegenstand der Verhandlung der Mitglieder der epistemischen Regime ist: Die Abwesenheit eines automatischen Installationskripts wird in Arch diskutiert und es steht Usern frei, eines zu generieren – wenn sie es können. Ebenso kann man sehen, wie bei Debian die Substitution des Administrator-Accounts durch Administratorrechte des User-Accounts (*sudo*) sowie die Bereitstellung von (inoffiziellen) Installationsmedien mit proprietären Treibern in den Installationsprozess eingeflossen sind. Dies sind Zugeständnisse an Laien, die in den letzten Jahren – man könnte sagen: seit der Gründung von Ubuntu – verstärkt sichtbar geworden sind und deren Bedürfnisse bei der Umsetzung stärker in Betracht gezogen wurden als noch wenige Jahre zuvor.

Im nächsten Kapitel widme ich mich nun den Ergebnissen meiner Analyse und der Beantwortung der Forschungsfragen der Arbeit. Anschließend diskutiere ich weitere Folgerungen in Bezug auf Rolle und Wirkung der epistemischen Regime sowie die Rolle von Expert*innen und Laien, Entwickler*innen und Usern und die differenzierten Abstufungen dazwischen.

Teil IV

Normative Konfigurationen

10 Zusammenfassung der Ergebnisse

In den vorangegangenen Kapiteln habe ich die normativen Konfigurationen der epistemischen Regime der Vergleichsfälle anhand der einzelnen Variablen betrachtet und anschließend die normativen Einschreibungen der epistemischen Regime in ihre Wissensprodukte am Beispiel der jeweiligen Installationsskripte analysiert. Nun werde ich die Ergebnisse zusammenfassen, um die Fragen nach der

1. Prägung der epistemischen Regime durch die normativen Vorstellungen ihrer Mitglieder und die
2. Auswirkungen der Konfiguration des Regimes auf das Wissensprodukt zu beantworten.

Dafür werde ich zusammenfassend beschreiben, wie in den spezifischen Fällen die analysierten Variablen zusammenspielen und welche Wechselwirkungen zwischen Wissensprodukt und Regime zu beobachten sind.

Im folgenden Kapitel werde ich die Folgerungen für die angewendeten Theorien ausarbeiten. Dabei diskutiere ich, was sich aus den Ergebnissen für die Frage des Verhältnisses von Expert*innen und Laien in FLOSS-Communities ableiten lässt und wo sich hierbei Ansatzpunkte für zukünftige Studien ergeben.

10.1 Spezifische (normative) epistemische Regime

Die freien Lizenzen der FLOSS-Projekte sind zunächst eine rechtliche Innovation, die ein grundsätzliches Recht einräumt, den abstrakten Quelltext der Software zu beziehen und nach eigenen Vorstellungen zu modifizieren. Für eine effiziente Zusammenarbeit ist es aber zielführend, die vielfachen Beiträge aus der Community zentral zu integrieren. Nur so ist gewährleistet, dass alle von den Veränderungen profitieren und diese nicht einzeln immer wieder für jede neue Version einpflegen müssen. Die Lizenz regelt aber nicht den Modus der Integration und der Zusammenarbeit, sondern hier ergeben sich je Fall spezifische Organisationsformen, die ich im Kapitel 8 ausgearbeitet habe. Dabei habe ich ein besonderes Augenmerk auf die Rolle von Laien

innerhalb der epistemischen Regime gelegt und die Grenzziehung zwischen den Expert*innen und Laien und zwischen Beitragenden und Nutzer*innen betrachtet.

Die epistemischen Regime der Communities regulieren dabei nicht nur die Beitragsmodalitäten für ihre User und Entwickler*innen, sondern erbringen durch die Einschreibung ihrer Werte in die Software eine Selektionsleistung, die User für potenzielle Beiträge für die Community anspricht oder ausschließt.

Die Auswahl der Beitragenden geschieht im Bereich der Entwicklung des Codes stark formalisiert über die Vergabe von Schreibrechten in das Software-Repository. In den erweiterten Beitragsmöglichkeiten der Integration von inkrementellen Änderungsvorschlägen (Patches) und im Bereich des Feedbacks über Bug Reports und der Arbeit an einer gemeinsamen Knowledge Base funktioniert die Selektion aber weniger formell. Hier werden die Funktionalitäten, Anwendungsmöglichkeiten und verschiedene Konfigurationen dokumentiert. Anders als bei der Integration von Code-Beiträgen ist die Selektion der Beiträge hierbei nicht einem direkten Peer-Review unterworfen. Vielmehr findet hier die Qualitätskontrolle interaktiv durch mehrere Diskussionsbeiträge statt (z.B. in Foren oder in Bugtrackern) oder durch wiederholtes Editieren und Korrigieren eines Textes (z.B. in einem Wiki).

Die epistemischen Regime sind also durch Wertsetzungen ihrer Mitglieder konfiguriert, die sich in den Strukturen und Verfahren der Community vergegenständlichen und damit zurückwirken auf die Selektion der formalen Mitglieder und die Rahmung der Beitragsmöglichkeiten. Über die Beiträge wird wiederum die Software definiert und so werden die Werte der Community in das gemeinsame Wissensprodukt eingeschrieben. Schließlich führt die Beschaffenheit der Software zu einer Adressierung von bestimmten User-Typen und erfüllt damit wiederum eine Selektionsleistung bezüglich der Mitglieder und potenziellen Beitragenden der Community.

Um die Wirkungsweise der Regime zu veranschaulichen, werde ich diese nun fallspezifisch zusammenfassen und das Zusammenspiel der Variablen herausarbeiten. Dabei werde ich die wesentlichen Elemente beleuchten und kontrastieren, um die Konturen stärker herauszuarbeiten.

10.1.1 Usability-Regime: Institutionalisierte Inklusion

Das epistemische Regime von Ubuntu Linux verkörpert eine institutionalisierte Inklusion von Laien, die von Anfang an auf die Einbindung von möglichst vielen Usern ausgelegt ist. Abbildung 10.1 zeigt die Ausprägungen der verschiedenen Dimensionen und das Zusammenspiel der Variablen.

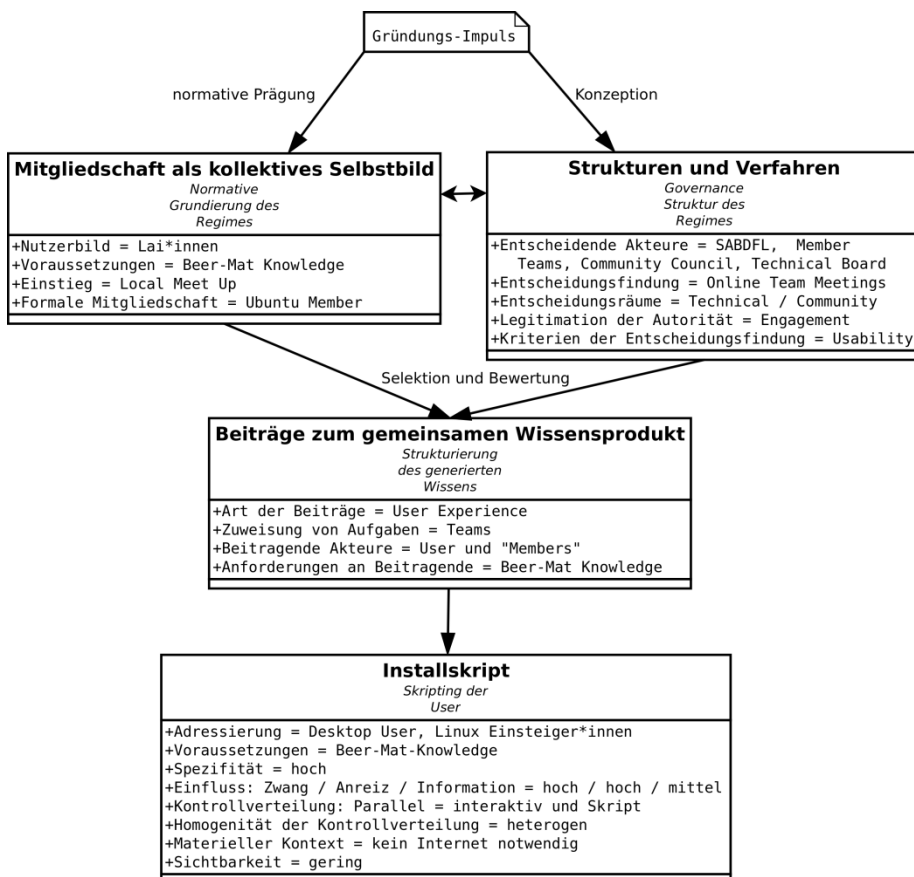


Abb. 10.1 Epistemisches Regime Ubuntu

Bemerkenswerterweise ging der Gründung von Ubuntu ein dezidiertes Planungstreffen voraus, in dem der Gründer Marc Shuttleworth zusammen mit einer Gruppe von Entwickler*innen aus anderen Projekten überlegte, wie ein „besseres“ Betriebssystem aussehen könnte und welche Community dieses System bauen würde (vgl. Hill u. a. 2006: 2). Nicht zuletzt wegen der großen Menge von unterstützter Software wurde Ubuntu als Fork von Debian

konzipiert.¹⁵⁰ Shuttleworth bezahlte den Entwickler*innen ein halbes Jahr Gehalt, um in dieser Zeit dann die gemeinsamen Ideen und Ansprüche umzusetzen. Unter den gesteckten Zielen war von Anfang an Nutzerfreundlichkeit zentral und eine starke Rolle der Community und Beitragsmöglichkeiten für User – “in whatever way they could” (ebd.: 10) – wurden anvisiert.

Als Nutzerbild werden bei Ubuntu ganz klar Nutzer*innen ohne technischen Hintergrund adressiert – Laien, die allenfalls ein schematisches Verständnis von Computern haben. Folglich ist für eine Mitgliedschaft in der Community auch lediglich *Beer-Mat Knowledge* notwendig. Explizit ist es erklärtes Ziel, Nutzer*innen des marktbeherrschenden Microsoft Windows zu erreichen und ein System anzubieten, das jeder User ohne tiefere Kenntnisse einrichten und verwenden kann. In diesem Nutzerbild finden sich auch Entwickler*innen wieder, die zwar die technische Expertise hätten, um am Betriebssystem zu basteln, aber sich auf ihre spezifischen Anwendungen konzentrieren wollen und sich nicht weiter mit dem System „herumschlagen“ möchten.

In die Struktur der Projektorganisation ist darüber hinaus Integration von Usern und Community-Management eingeschrieben. Es gibt eine dezidierte formale Mitglieder-Rolle für engagierte User – mit Mitgliedsurkunde –, die sich jenseits von technischen Beiträgen für die Community engagieren, sowie ein formales Gremium für Community-Angelegenheiten, den *Community Council*. Ferner gibt es ein dezidiertes Team bezahlter Mitarbeiter*innen, die aktiv das soziale Miteinander pflegen.

Die Community wird hierbei als wertvolle Ressource für die Produktentwicklung begriffen und die Governance-Struktur daraufhin entwickelt. Laien liefern dabei ihre subjektive Perspektive auf die einfache Nutzbarkeit des Systems. Strukturell wird das Zugehörigkeitsgefühl von Usern, die nicht auf der Code-Ebene beitragen, durch eine formale Rolle als *Ubuntu Member* unterstützt, und zugleich werden deren Beiträge über die organisationale Anerkennung durch Mitwirkungsrechte aufgewertet.

Diese formelle Unterstützung sozialen Engagements und der erfolgreiche Aufbau einer Identifikation mit dem gemeinsamen Projekt schlägt sich nieder in lokalen Anwendertreffen, in denen sich User gegenseitig unterstützen und das gemeinsame Produkt bewerben und die wiederum Anknüpfungspunkte für neue Mitglieder darstellen. Hier fruchtet das Branding als „Linux for

150 Als Fan von Debian hatte Shuttleworth vorher gründlich überlegt, ob er, statt Debian zu „forken“, es auch reformieren könnte (vgl. Hill u. a. 2006: 9).

Human Beings“. Schließlich führt die Anerkennung und Integration von Laien zu einer verstärkten Übersetzung technischen Wissens für einfache Nutzer*innen. Deren Fragen und Antworten prägen durch ihre Beiträge aus reiner User-Sicht das produzierte Wissen stark und tragen schließlich zu Dokumentationen bei, die auch von Laien verstanden werden können. Durch die Akzeptanz auch unausgereifter Bug Reports, die in einem weiteren Schritt von „Semi-Experten“ triagiert werden – also für die Entwickler*innen aufbereitet –, erhöht sich idealiter das generelle Feedback und durch die explizit genannte Beitragsform des Bug Triaging gibt es Beitragsmöglichkeiten für User verschiedener Expertise-Grade.

Sowohl für die Mitgliedschaft als auch für valide Beiträge und schließlich für die erfolgreiche Installation ist lediglich ein schematisches Verständnis der Software notwendig. Das bedeutet allerdings nicht, dass die Schwelle für Entwickler*innen, die den Code pflegen, niedrig ist; hier gelten vergleichbare Kriterien wie in den anderen betrachteten Fällen. Vielmehr wird zusätzlich Engagement jenseits technischer Expertise motiviert und nach Möglichkeit eingebunden. Zentrale Kriterien sind dabei die Verständlichkeit und einfache Bedienbarkeit der Oberfläche des Systems.

Dies zeigt sich sehr deutlich in der Betrachtung des Installationskriptes als grundlegende Software-Komponente. Zunächst bietet der Installer ein Live-System zum Testen an, mit dem der User sich von den Qualitäten des neuen Systems überzeugen kann, und gibt somit Anreize, die Installation zu starten. Der Installationsprozess selbst ist einfach gestaltet und verlangt relativ wenig Vorwissen. Er bietet nur wenige Optionen, zwischen denen User wählen müssen, und es gibt klare Empfehlungen, die Standardkonfiguration zu verwenden. Das System ist außerdem sehr spezifisch auf den üblichen Arbeitsplatzrechner abgestimmt und eine breite Palette von Standard-Anwendungen wird vorinstalliert. Daraus ergibt sich ebenfalls eine klare Adressierung von „Normal-Nutzer*innen“.

Das Skript hat also eine enge Benutzerführung und ist außerdem auf eine niederschwellige Installation angelegt. Zu Beginn des Programms wird die Festplatte vorbereitet und der zeitaufwendige Kopiervorgang gestartet. Somit ist die Entscheidung für das System schon gefällt, während weitere Details der Konfiguration abgefragt werden. Einerseits wird der User dadurch animiert, die Installation nicht mehr abzubrechen – anders, als wenn erst eine Reihe von Konfigurationsfragen kommt –, und andererseits muss der User anschließend nicht mehr lange warten, bis das System betriebsbereit ist. Über die restliche verbleibende Zeit wird der User über Screenshots und Beschrei-

bungstexte schon in das System eingeführt, der User wird also durch die Vorschau schon neugierig gemacht und für die Installation belohnt.

Das Installationsskript verlangt dem User also wenige Entscheidungen ab und macht die Installation über Voreinstellungen und die grafische Gestaltung möglichst angenehm. Damit werden die Hürden für eine Installation so niedrig wie nur möglich gehalten, um möglichst viele User zu erreichen und damit auch potenzielle Beitragende für die Community zu gewinnen. Das Skript fügt sich also ein in die Governance-Struktur der Community, leistet einen eigenen Beitrag zur Integration von einfachen Nutzer*innen und spiegelt damit den Gründungsimpuls des Projekts wider.

10.1.2 Stability Regime: Demokratie qualifizierter Entwickler*innen

Debian bildet technisch gesehen den Ausgangspunkt des darauf basierenden Ubuntu, unterscheidet sich aber fundamental in der Organisationsstruktur und bringt auch ein anderes Nutzerbild mit sich. Das epistemische Regime Debians mit den spezifischen Ausprägungen ist in Abbildung 10.2 schematisch dargestellt.

Debian hat eine besondere Reputation in der FLOSS-Community, da das Projekt in vier Bereichen Maßstäbe setzt. In technischer Hinsicht besticht Debian durch die hohe Anzahl von angebotenen Softwarepaketen (> 25.000) und die vielen verschiedenen unterstützten Hardware-Architekturen (s. auch S. 129). Darüber hinaus ist Debian eines der größten FLOSS-Projekte und bekannt für das Bekenntnis zu den ethischen Prinzipien von Freier Software, die im *Social Contract* und den *Debian Free Software Guidelines* festgehalten sind (vgl. Coleman 2013: 129). Debian setzt also Maßstäbe in der FLOSS-Welt. Die Größe der Community hängt zusammen mit dem Gründungsimpuls, den Eliten-zentrierten Projekten eine Organisationsform entgegenzusetzen, die allen (technisch) Beitragenden eine Mitgliedschaft ermöglicht – vorausgesetzt, sie erfüllen die notwendigen Voraussetzungen. Die verfassten Prinzipien bilden dabei einen wichtigen normativen Bezugspunkt.

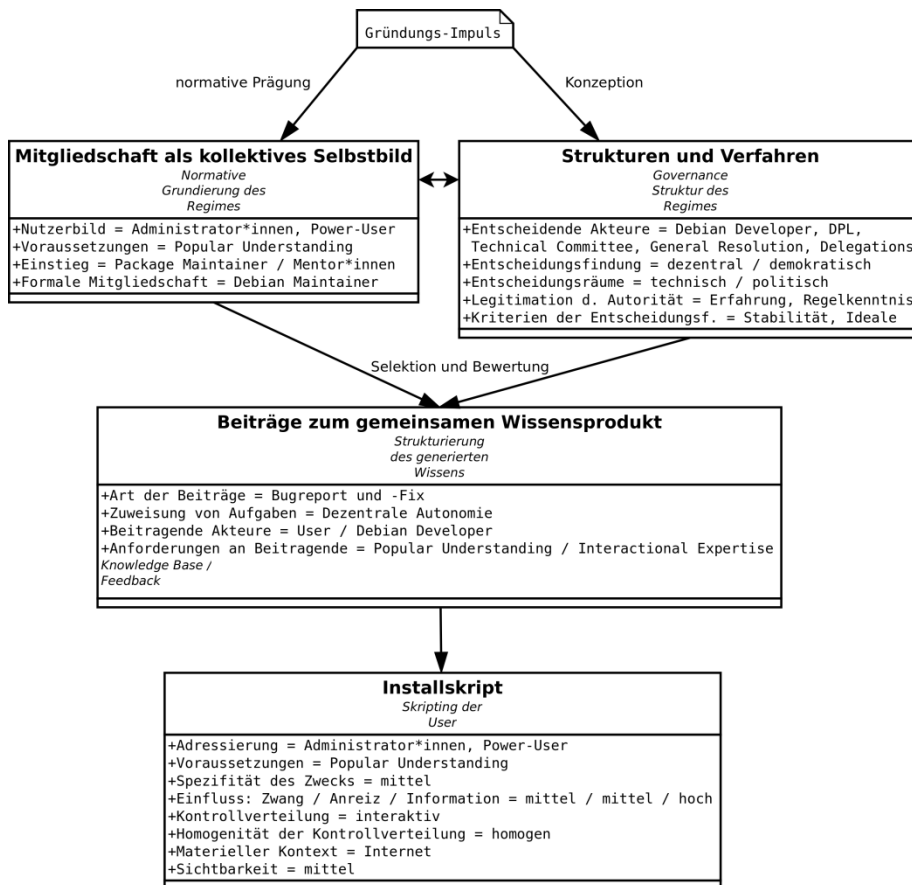


Abb. 10.2 Epistemisches Regime Debian

Auch wenn durch die freien Lizenzen formal jede*r Einzelne das Recht hat, den Softwarecode zu verändern, ist doch jede*r Entwickler*in darauf angewiesen, dass die Veränderungen auch in die gemeinsame Code-Basis, das Repository, aufgenommen werden. Andernfalls muss die Veränderung nach jedem Update als Patch wieder individuell eingepflegt werden und steht anderen Usern nicht zur Verfügung. Dies ist in Anbetracht der investierten Zeit und vor dem Hintergrund einer Philosophie des Teilens für die Beitragenden unbefriedigend. Ein wichtiger Aspekt bei der Gründung von Debian war daher die Idee, möglichst allen beitragenden Entwickler*innen die Möglichkeit zu geben, Teil der Produktionsgemeinschaft zu werden. Somit wollte der Gründer Ian Murdock einer elitären Willkür von Core-Entwickler*innen

entgegenwirken und den Beitragenden ermöglichen, sich das Projekt zu eigen zu machen und auch an Entscheidungen mitzuwirken.

Das Paketmanagementsystem von Debian war eine grundlegende Innovation, wodurch die vielen einzelnen Entwickler*innen unabhängig an den von ihnen betreuten Paketen arbeiten konnten. Dies sollte einen Wendepunkt darstellen in der Geschichte der gemeinsamen Systementwicklung, weil dadurch Transparenz, Zugänglichkeit und Offenheit ermöglicht wurden und schließlich Partizipation motiviert und erleichtert wurde (vgl. ebd.: 129 f.).

Folglich wurde die Struktur der Community im Hinblick auf transparente Aufnahmekriterien gestaltet, um eine willkürliche Selektion durch eine selbsternannte Elite zu unterbinden. Mentorenschaft spielt dabei eine zentrale Rolle, um die Gemeinschaft durch soziale Interaktion zu erweitern und Einsteiger*innen zu sozialisieren. Die grundsätzliche Öffnung der Community für einen größeren Kreis von Mitgliedern erfordert für die Gewährleistung einer hohen Qualität der entwickelten Software einen hohen Anspruch an die Mitgliedschaft, der durch ein relativ schwieriges, zweistufiges Bewerbungsverfahren sichergestellt wird: Bewerber*innen müssen darin nachweisen, dass sie Erfahrungen und eine gute Qualifikation haben, die Regeln der Community kennen und uneigennützte Ziele und Visionen für das Kollektiv verfolgen.

Der Einstieg in die Mitgliedschaft ergibt sich üblicherweise über die Zusammenarbeit mit einem oder einer Entwickler*in an einem von ihm oder ihr betreuten Paket. Die formale Mitgliedschaft ergibt sich also stets über das Beitragen und vermittelt durch die betreffenden Maintainer, die als Mentor*innen für die neuen potenziellen Mitglieder erscheinen.

Der Status des *Debian Developer* ist eine starke Rolle mit weitgehenden Rechten in der Community, denn jeder ist autonom in seiner Nische, in der Betreuung seiner Pakete, und hat mit allen anderen eine gleichwertige Stimme für politische Entscheidungen (*General Resolution*) sowie bei der Wahl des *Debian Project Leader*, der eher eine repräsentative Funktion innehat und die Community „moralisch“ unterstützt. *Debian Developer* können auch Teil des *Technical Committee* werden, das kritische Entscheidungen in technischen Angelegenheiten¹⁵¹ berät und entscheidet, wo anderweitig keine Einigung erreicht werden kann.

151 An dieser Stelle sei nochmals angemerkt, dass die Unterscheidung von Entscheidungen technischer und politischer Natur in der Praxis nicht immer trennscharf zu ziehen ist.

Bei der Einarbeitung in die Strukturen und Policies der Communities und das Sammeln von Erfahrungen hilft das Konzept der Mentorenschaft, denn auf die Befolgung der Regeln für die Kontribution wird hohes Wert gelegt. In Bezug auf die Beiträge für das Projekt herrscht in der Bewertung der Community klassischerweise ein eher technisches Verständnis. Dies zeigt sich beispielsweise an der Vernachlässigung der Dokumentation (die Funktion leitet sich aus dem Sourcecode ab), aber auch in vergleichsweise starken Anforderungen an Beiträge im Bereich des Feedbacks (*Interactional Expertise*).

Im Kontrast zu Ubuntu besteht für Beiträge auf der Feedback-Ebene eine höhere Anforderung an die Beitragenden. Die hohe Schwelle für Beiträge korreliert mit einer geringeren Beteiligung von Beitragenden mit weniger technischer Expertise. Die Vermutung liegt nahe, dass daher User mit tieferer (technischer) Expertise sich nach der Einarbeitung mit ihrem höheren Wissensniveau zufriedengeben und wenig Motivation haben, dieses Wissen in eine einfachere Dokumentation fließen zu lassen, die für weniger erfahrene Nutzer*innen leichter verständlich ist.

Das wiederum deckt sich mit einem Nutzerbild von Administrator*innen und Power-Usern, die technisch versiert sind und im Sinne des “when the code is public, RTFM is the proper answer” (S. 129) durch die Auseinandersetzung mit der Software zu Expert*innen werden und verstehen, dies für sich nutzbar zu machen. Grundlegende Kenntnisse (*Popular Understanding*) erscheinen vor diesem Hintergrund als gegeben, und ausbaufähig in Richtung einer *Interactional Expertise*. Dies spiegelt sich schließlich auch im Installationskript wider, das trotz vorgefertigter Konfigurationsmenüs technische Kenntnisse impliziert (vgl. Abschnitt 9.2).

Im Vergleich zur engen Benutzerführung bei der Ubuntu-Installation haben die User weitreichende Konfigurationsmöglichkeiten und wenige Vorgaben; der Assistent kann jederzeit verlassen werden, um einzelne Schritte anzuwählen. Angeboten wird auch die Option, eine Kommandozeile zu starten, um eigenhändig mit der Eingabe von Kommandos Konfigurationen vorzunehmen. Es herrscht hier also deutlich weniger Zwang und mehr Wahlfreiheit im Skript. Die Wahrnehmung dieser Freiheiten geht aber einher mit notwendigem Wissen über die Einstellungsmöglichkeiten.

Die Debian Community ist also generell eher auf erfahrenere User ausgerichtet. Wenngleich an manchen Stellen ein Schritt auf weniger erfahrene User zugegangen wird – beispielsweise durch Anfänger-Optionen bei der Installation –, konzentriert sich die Organisation der Community eher auf die

technischen Kontributoren. Diese können jedoch weitreichende Rechte in der Community erhalten, wodurch eine große Community entstand, die über die sonst üblichen kleinen Core-Developer-Eliten weit hinausgeht und einem großen Kreis von Mitgliedern weitreichende Rechte in der Community gibt. Diese Ausweitung erfordert klare normative Referenzpunkte, wie den *Social Contract*, und wird stabilisiert durch ein umfassendes Bewerbungsverfahren.

10.1.3 Minimales Eliten-Regime

Ganz im Gegensatz zu Ubuntu wendet sich Arch ganz explizit ab von der Usability-Idee und begründet stattdessen einen Begriff der *User Centricity*. Damit gemeint ist eine Fokussierung auf die User, die auch aktiv zum Projekt beitragen. Im Gegensatz zu Debian gibt es aber nur eine kleine Gruppe von *Core-Developern*, die die Richtung des Projekts vorgeben. Zentrales Leitmotiv ist das Prinzip „Keep It Simple, Stupid“ – eine technische Simplizität, die einfach zu warten sein soll, anstelle der Verbergung technischer Zusammenhänge durch zusätzliche Komplexität grafischer Oberflächen beispielsweise. Dementsprechend werden User angehalten, ein technisches Grundwissen zu erlangen, um das System besser warten zu können, ohne dafür einfache grafische Oberflächen bereitzustellen.

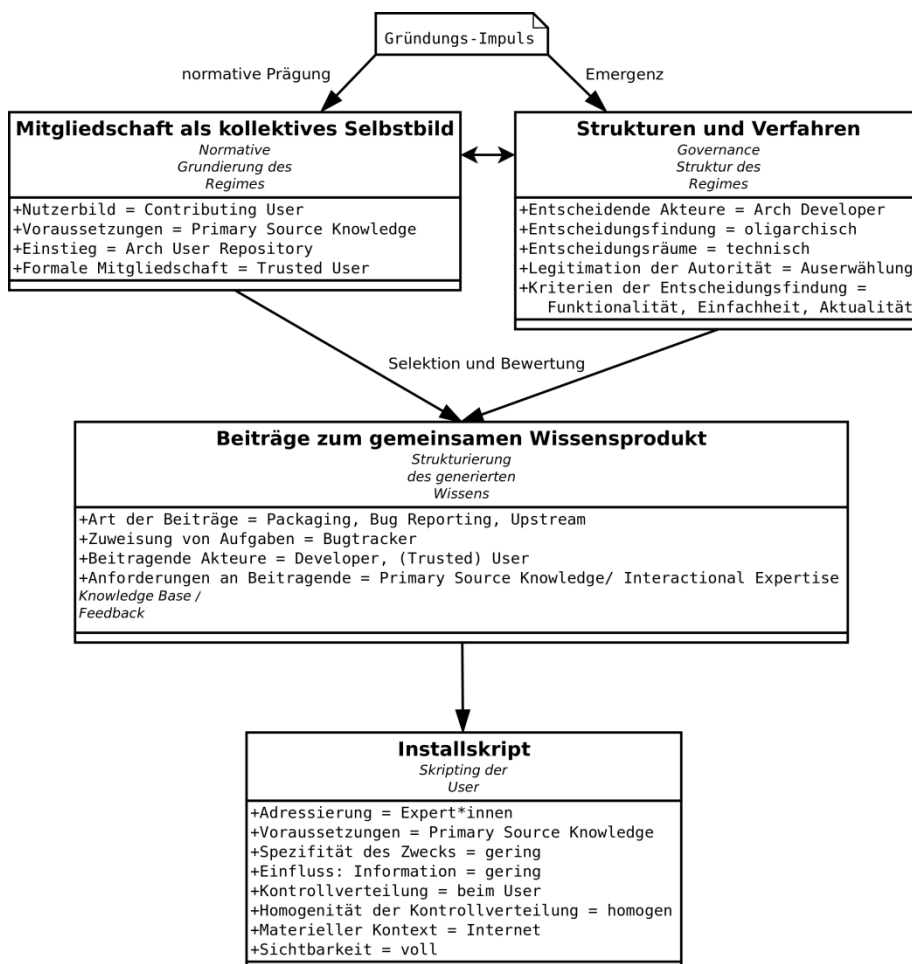


Abb. 10.3 Epistemisches Regime Arch

Das Prinzip der *User Centricity* prägt das Nutzerbild der Community. Um Arch zu benutzen, wird erwartet, dass man die Fähigkeiten für das aktive Beitragen für die Community mitbringt oder zumindest die Bereitschaft, die einschlägige Dokumentation zu lesen (*Primary Source Knowledge*, RTFM). Bevor man eine formale Rolle als Trusted User anstrebt, ist es zielführend, seine Fertigkeiten im *Arch User Repository* unter Beweis zu stellen. Dadurch ist der Einstieg in die Kontribution also schon direkt auf der Entwickler-Ebene angesiedelt.

Zentrale Akteure in der Organisationsstruktur sind die *Arch Developer*. Sie sprechen anstehende Entscheidungen untereinander ab. Da es sich um

eine recht kleine, homogene Gruppe handelt, gibt es keine formalen Prozesse. Gibt es Bedarf an personeller Unterstützung, werden Kandidat*innen aus der Community direkt von den Entwickler*innen angesprochen. Dadurch wird gewährleistet, dass die Gruppe relativ homogen bleibt. Die Entscheidungskriterien sind aufgrund des minimalistischen Credo der Community auch klar gesteckt: Aktualität, Funktionalität und Einfachheit bilden einen starken Konsens technischer Entwicklung. Das zweite Entscheidungsorgan der Trusted User verwaltet lediglich die weniger relevanten Anwendungspakete im *Community-Repository* und kümmert sich um das *Arch User Repository*.

Insgesamt ist Arch ein relativ kleines Projekt, bei dem die informelle Koordination offenbar recht gut funktioniert, zumal die Distribution auch nur eine minimale Infrastruktur anbietet und die Upstream-Software nahezu unmodifiziert in ihren Repositories zur Verfügung stellt. Dadurch sind wenige, klare Kriterien wie Funktionalität, Minimalismus und Aktualität zielführend und es wird relativ wenig Meta-Organisation benötigt.

Entsprechend der klaren technischen Ausrichtung besteht auch eine große Einigkeit darüber, dass Bug Reports mit hinreichend Informationen ausgestattet sein müssen und nach Möglichkeit Bugs idealerweise im Upstream reported und behoben werden sollen. Hierzu sind dann auch keine spezifischen Rechte für das Repository der Distribution nötig, sondern sobald die Änderung dort übernommen wird, fließen die Änderungen in die hochaktuelle Arch-Distribution. Stärker als bei den anderen beiden Fällen werden hier alle User als Beitragende betrachtet. Dabei wird ein vergleichsweise hohes Maß an Expertise vorausgesetzt. Für das Beitragen zum Bereich der Knowledge Base ist mindestens *Primary Source Knowledge* nötig, im Bereich von Feedback ist *Interactional Expertise* essenziell, um den Anforderungen der Community gerecht zu werden. Schließlich ist die Schwelle zur Nutzung der Software schon recht hoch (was sich bei der Installation zeigt) und das Beisteuern von AUR-Softwarepaketen wird explizit als User-Beitrag verstanden. User haben also dezidierte Entwickleraufgaben. Die zentrale Rolle für Entscheidungen bleibt den *Arch Developern* vorbehalten, zu deren Kreis nur explizit *ausgewählte* Beitragende gehören.

Aufgrund der hohen Anforderungen an die User, die ja explizit auch als Beitragende adressiert werden, bleibt die Installationssoftware ohne jegliche Automatisierung. Vielmehr beinhaltet das „Skript“ eine Übersicht der notwendigen Schritte und Befehle, für deren Ausführungen ein*e Unkundige*r sich die einschlägige Dokumentation selbst erarbeiten muss. Somit hat der

User völlige Freiheit, wie er das System installieren möchte, und qualifiziert sich durch die Installation schließlich für eine aktiv beitragende Rolle in der Community, da eine Kenntnis der Dokumentation für die erfolgreiche Installation erforderlich ist.

Die Fokussierung auf technisch beitragende User zeigt sich bei Arch sehr stark in der betrachteten Installationsroutine. Der User muss zum Experten werden, um überhaupt das System zu installieren. Die Governance-Struktur gibt allen Beitragenden zwar die Möglichkeit, eigene Software-Pakete im User Repository zu pflegen, die richtungsweisenden Entscheidungen werden aber in einem kleinen Kreis von Core-Developern getroffen. Dies wird aber von den Befragten nicht problematisiert, sondern es scheint einen Konsens zu geben, der die technische Autorität der erfahrenen Elite akzeptiert. Die Gestaltung der Software orientiert sich also stark an technischen, beitragenden Usern, die strategische Organisation bleibt aber einer kleinen Gruppe vorbehalten. Eine strukturelle Partizipation der Beitragenden wird also nicht für nötig erachtet, das politische Element technischer Entscheidung bleibt in diesem Verständnis ausgeblendet.

10.2 Das Zusammenspiel der Variablen epistemischer Regime

In den betrachteten Regimen zeigt sich ein Zusammenspiel der analytischen Variablen der *Strukturen und Verfahren* der Organisation, der Gestaltung der *Mitgliedschaft* und der Praktiken des *Beitragens zum gemeinsamen Wissensprodukt*. Schließlich wirken sich die Regime aus auf die produzierte Software – und diese wirkt sich selektiv aus auf die Mitgliedschaft.

Wie im vorausgehenden Abschnitt herausgearbeitet wurde, wurden die Strukturen und Verfahren der Community mit bestimmten Intentionen entworfen und wirken sich auf die Organisation des Kollektivs aus. Ubuntu und Debian haben bewusst entworfene formale Prozesse, die die Werte der Integration von technisch Beitragenden (Debian) und die Integration jeglicher (nicht technischer) Beitragender (Ubuntu) festschreiben. Diese sind Gegenstand der Verhandlung, werden also von den Mitgliedern zumindest diskursiv beeinflusst, und entwickeln sich so im Laufe der Zeit weiter. Bei Arch wird

auf eine komplexe formale Organisation verzichtet, hier spiegelt sich sozial das Credo des technischen KISS-Prinzips wider.

Durch die formale Struktur werden die Kriterien und Bedingungen der Mitgliedschaft und bestimmte Einstiegsfade in die Kontribution definiert und der Rahmen für die Selektion und Bewertung der Beiträge gesteckt. Bei beidem spielen aber auch starke informelle Elemente eine bedeutende Rolle.

Der informelle Anteil der Mitgliedschaft wird einerseits interaktiv durch die kommunikativen Praktiken in der Wissensproduktion geprägt, wo Beiträge als valide akzeptiert oder als unzureichend abgewiesen werden. Andererseits wirkt die Gestaltung der Software selbst auf das kollektive Selbstbild zurück, da die Beschaffenheit der Software die Präferenzen der Community widerspiegelt oder eben Mitglieder selektiert, die diese Präferenzen teilen.

Die Praxis des Beitragens steht in einer Wechselbeziehung mit der Mitgliedschaft und dem damit verbundenen kollektiven Selbstbild, da die Mitglieder die gemeinsame Interaktion strukturieren. Durch formale Rollen und Rechte der Bearbeitung des gemeinsamen Wissenspools, aber auch formalisierte Regeln wie den *Code of Conduct*, wird das Beitragen durch die formalen Strukturen gerahmt.

Schließlich formt die interaktive Aushandlung in der Selektion und Bewertung von Beiträgen das Wissensprodukt. Dieses ist zugleich Produkt und Ausgangspunkt der gemeinsamen Produktion und hat somit einen direkten Einfluss auf die Aneignung von Wissen in der Community. Sowohl die Beschaffenheit der Dokumentation als gemeinsame Wissensbasis als auch die Beschaffenheit der Software selektiert User und Beitragende und beeinflusst so die Zusammensetzung der Mitglieder.

Im Folgenden werde ich die einzelnen Variablen nochmals gesondert betrachten und die verschiedenen Ausprägungen der betrachteten Communities kontrastieren.

10.2.1 Mitgliedschaft ist Selektionsmechanismus und Identitätsmoment

Die Mitgliedschaft der betrachteten Communities erfüllt mehrere Funktionen. Sie selektiert User als potenziell Beitragende und stiftet Identität und damit Verpflichtung gegenüber den Zielen, Regeln und Idealen der Gemeinschaft. Schließlich schafft sie als formale Rolle eine explizite Anerkennung des Engagements für die Community.

Dies geschieht einerseits über die Nutzung der Software durch die Einschreibung der Werte der Community. Die Software spiegelt durch Design, Funktionalität und Vorkonfiguration sowie die Gestaltung der Konfigurationsmöglichkeiten gewisse Werte wider. Die Distribution bildet hierbei das Tor zur Welt der FLOSS und bettet die unzähligen Programme auf spezifische Art und Weise ein. Die spezifische Beschaffenheit der Software impliziert bestimmte Kenntnisse im Umgang mit dem Betriebssystem und impliziert (ermöglicht oder beschränkt) somit die Nutzung für bestimmte User-typen.

Die Nutzung ist aber nur ein erster Schritt für die Mitgliedschaft einer Community. An die Nutzung schließen sich vielfältige Partizipationsmöglichkeiten an:

- die Gestaltung des Community-spezifischen Wissens („Knowledge Base“);
- Feedback-Möglichkeiten bezüglich der Beschaffenheit der Software (einschließlich Rückmeldung über Fehler sowie Wünsche über neue Funktionalitäten);
- Beiträge von Softwarecode in Form von kleinen Änderungsvorschlägen („Patches“) über die Mitarbeit an der Anpassung von Software für die Distribution („Packaging“), bis hin zur
- Mitarbeit an den spezifischen Software-Projekten außerhalb der Distribution („Upstream“).

An die Mitgliedschaft werden unterschiedliche Voraussetzungen gestellt – und zwar in Form der von der Community erwarteten aktiven Beteiligung am System und darüber hinaus in Form der Voraussetzungen, die an die Beiträge gestellt werden, die von Usern für die Community erbracht werden.

Diese Voraussetzungen finden sich in institutionalisierten Einstiegsangeboten, über die User in eine aktivere Community-Rolle „wachsen“ können, und schließlich in den Anforderungen für eine formale Mitgliedschaft. Die formale Mitgliedschaft wirkt dann als explizite Anerkennung des spezifischen Mitglieds, seinem Engagement und seiner Leistungen und stiftet durch die explizite Zugehörigkeit und Anerkennung Identität und somit Motivation. Außerdem wirkt das anerkannte Mitglied als Repräsentant der Werte der Community und als Vorbild einer aktiven Community-Rolle, die das Gesamtprojekt bereichert und voranbringt.

Auf diese Weise werden die Werte der Community festgeschrieben und trotz der durchlässigen Grenzen der Community stabilisiert.

10.2.2 Strukturen und Verfahren als formale zentrale Governance-Struktur

In den Strukturen sind die Voraussetzungen für die formale Mitgliedschaft in die Bewerbungsverfahren eingeschrieben. Häufig wird darin sichergestellt, dass andere Mitglieder, die das Verfahren schon durchlaufen haben, die neuen Bewerber anhand ihrer Kompetenzen und Beiträge bewerten. Auf diese Weise wird sichergestellt, dass die neuen Mitglieder gewisse Eigenschaften mitbringen und bestimmte Werte der Gemeinschaft teilen. In Ubuntu spielt hierbei der *Code of Conduct* eine wichtige Rolle, der den Umgang miteinander regelt; in Debian gibt es einen Sozialvertrag, der für das Projekt die gesellschaftliche Bedeutung von Freier Software als Community-Ziel fest schreibt. Die Kenntnis und Unterstützung derartiger Regeln spielen für eine erfolgreiche Bewerbung als Mitglied eine wichtige Rolle.

Die Rollen in der Community sind außerdem häufig nach verschiedenen Befugnissen gestaffelt, an die wiederum Voraussetzungen geknüpft sind. Die Befugnisse unterscheiden sich in der Berechtigung, bei bestimmten Entscheidungsprozessen eine Stimme abzugeben. Bei Ubuntu wird beispielsweise differenziert zwischen *Ubuntu Members* und *Ubuntu Developers*, die jeweils bei der Besetzung des Gremiums für Community-Angelegenheiten oder aber bei der Besetzung des Gremiums für technische Angelegenheiten mitentscheiden dürfen. In den unterschiedlichen Communities gibt es dementsprechend auch Gremien, die für Entscheidungsprozesse gewisse Kriterien institutionalisieren. In Ubuntu gibt es einzig ein Gremium für Community-Angelegenheiten, in Arch gibt es kein derartiges Gremium jenseits der Gruppen *Arch Developer* und *Trusted User*. Außerdem gibt es eine Differenzierung verschiedener Bereiche der Software-Repositories, für deren Zugriff verschiedene Rollen qualifizieren, beispielsweise Kernbereiche, auf die nur langjährige, erfahrene Entwickler*innen zugreifen dürfen, und Community-Bereiche, in denen Entwickler*innen sich durch gute Arbeit bewähren können, bis hin zu einem individuell zugreifbaren Bereich, in dem jeder User aktiv werden darf. Diese Bereiche spielen dabei unterschiedliche Rollen, wie in Abschnitt 8.2 herausgearbeitet wurde.

Der Modus der Entscheidungsfindung selbst unterscheidet sich wiederum je nach Gegenstand und Community. Neben einer Open-Source-typischen pragmatischen Koordination, in der sich diejenigen zu Wort melden, die am Thema bzw. am Code arbeiten, und die erfahreneren, „verdienten“ Mitglieder eine stärkere Autorität besitzen, gibt es auch Wahlverfahren, bei denen eine

große Anzahl von Mitgliedern wählt. Ein solches Verfahren stellt bei Debian die *General Resolution* dar, die aber explizit nur „politische“ Entscheidungen treffen soll, die nicht „technisch“ zu argumentieren sind. Andere Angelegenheiten werden eher über Gremien entschieden, die von der Community delegiert sind und damit durch ihre Position legitimiert werden, für das Projekt Entscheidungen zu treffen – beispielsweise Debians Technical Committee oder Ubuntu's *Technical Board*.

Schließlich spielt in der Delegation von Entscheidungen an bestimmte Mitglieder oder Gremien die Legitimation eine entscheidende Rolle. So werden im Ubuntu beispielsweise die Gremien vom Projektleiter Mark Shuttleworth ernannt, aber durch eine Bestätigung per Wahl von den Mitgliedern und Entwickler*innen bestätigt und somit legitimiert. Bei Debian wird der Projektleiter selbst jährlich gewählt und somit seine Position begründet. Bei Arch gibt es im Kontrast dazu keine Wahlen oder Bewerbungsverfahren für die Elite der *Arch Developer*. Sie sind einzig durch ihre Expertise und ihr Engagement legitimiert und erwählen eigenständig nach Bedarf neue Mitglieder der führenden Elite.

Auffallend ist hierbei, dass die Teile des gemeinsamen Wissens, die nicht funktional kritisch sind, also die Bereiche der Dokumentation (Knowledge Base) und Feedback (Bug Reports u. Ä.), kaum formal strukturiert sind, sondern hier die Selektion der Beiträge in der direkten Interaktion geschieht beziehungsweise durch eine generelle Selektion der Mitglieder. Somit wirken die impliziten und expliziten Voraussetzungen für die Nutzung der Software, den Einstieg in die Community und die Erlangung einer formalen Rolle zurück auf die Zusammensetzung der Mitglieder, die schließlich im Wesentlichen die Beiträge zum gemeinsamen Wissensprodukt generieren.

10.2.3 Die Selektion der Beiträge strukturiert das Wissensprodukt

Man kann im Wesentlichen zwei Arten von Beiträgen unterscheiden. Beiträge, die den Code der Software betreffen, sind formal stärker reguliert, da von der Art der Beiträge die Funktionalität der Software direkt betroffen ist (also der funktionale Teil des Wissensprodukts). Jede*r kann solche Beiträge anbieten, aber die Integration des Beitrags in das gemeinsame Repository obliegt einer Qualitätskontrolle durch Inhaber*innen der formalen Rolle: Die zuständigen Entwickler*innen integrieren die angebotenen Beiträge nach

einem Peer-Review und fordern von den Beitragenden bei Bedarf Nachbesserungen. In den betrachteten Fällen unterscheiden sich vor allem die formalen Rollen und die Prozesse, um diese Rollen zu erlangen. Bei Debian kann jede*r das umfangreiche und voraussetzungsvolle Bewerbungsverfahren durchlaufen, um ein vollwertiges Mitglied des gesamten Entwicklerteams zu werden (*Debian Developer*). Bei Ubuntu gibt es darüber hinaus auch explizite formale Nicht-Entwickler-Rollen (*Ubuntu Member*). Bei Arch schließlich gibt es kein Bewerbungsverfahren für den Status als Core-Developer (*Arch Developer*), sondern diese werden bei Bedarf ausgewählt. Lediglich zum Kreis der erweiterten Entwickler*innen kann man über ein Bewerbungsverfahren gelangen.

Beiträge, die nicht direkt die Funktion der Software betreffen, sind weniger stark formal reguliert. Dies betrifft insbesondere die Knowledge Base, den Bereich des Feedbacks über Dysfunktionalitäten (Bugs) und zusätzlich gewünschte Funktionen oder Funktionsänderungen. Das heißt, das Beitragen ist hier in der Regel nicht an eine formale Mitgliedschafts-Rolle gebunden, die an festgelegte Voraussetzungen geknüpft ist.

Im Gegensatz zu den formalisierten Bereichen des Beitragens von Softwarecode sind für das Beitragen zu den Bereichen Feedback und Knowledge Base auch keine Programmierkenntnisse erforderlich. In der damit verbundenen Auseinandersetzung mit der Technik ist aber zumindest eine Art von *Interactional Expertise* – im Sinne der Fähigkeit, mit den Entwickler*innen durch ein tieferes Verständnis der Funktionsweise des Systems auf Augenhöhe zu kommunizieren – hilfreich und in *manchen Communities auch notwendig*, um ernst- oder überhaupt wahrgenommen zu werden. Hierbei unterscheiden sich die betrachteten Fälle in den informellen Anforderungen an Beiträge. Während bei Arch unzureichende Fehler-Reports tendenziell ignoriert werden, wird bei Debian die Einhaltung der strikten Policies gefordert. Bei Ubuntu schließlich werden unzureichende Bug Reports im Rahmen des Bug Triaging mit den fehlenden Informationen angereichert.

Im letzten Fall ist also ein Beitrag mit geringen Kenntnissen durchaus legitim, während in den anderen beiden Fällen mehr oder weniger explizit ein tieferes Wissen gefordert wird. In Ubuntu führt so die Beteiligung von Usern ohne tiefere Expertise bei der Produktion von Anleitungen zur Nutzung der Software dazu, dass hier deren Perspektive mit einfließt, was die entstehenden Anleitungen und Dokumentationen für weniger erfahrene User leichter zugänglich macht.

Die Zuweisung von Aufgaben geschieht generell durch das Bekanntgeben von Aufgaben im Rahmen des genannten Feedbacks. Die Beitragenden können sich dann eigenständig aus der Liste von Verbesserungsmöglichkeiten passende Aufgaben aussuchen. Im Rahmen von formalen Rollen gibt es auch Zuweisungen und Zuständigkeiten, etwa durch Rollenbeschreibungen und definierte Aufgabenbereiche. Bestimmte definierte Gruppen von Mitgliedern sind üblicherweise für spezifische Teile der Software verantwortlich. Darüber hinaus werden in Debian und Ubuntu beitragswilligen Neulingen mitunter besonders einfache Aufgaben angeboten, um ihnen so den Einstieg zu erleichtern. Dies widerspricht der ursprünglichen Logik des „scratch your own itch“, entspricht aber dem Bedürfnis mancher Einsteiger*innen und dieses wird daher in den genannten Fällen bedient.

Begrenzt ist die Zuweisung von Aufgaben ganz grundsätzlich dadurch, dass die Mitglieder der Community völlig freiwillig mitarbeiten – mit Ausnahme der Mitarbeiter von Canonical, die in Ubuntu als Arbeitnehmer weisungsgebunden operieren.¹⁵²

10.2.4 Normative Prägungen der Regime

Die implizite, informelle und die explizite, formale Selektion der Beiträge strukturieren schließlich die Inhalte der Wissensproduktion. Durch die Akzeptanz der Beiträge von Usern verschiedener Erfahrungsstufen fließt unterschiedliches Feedback über deren Bedürfnisse in die Gestaltung der Software ein und durch deren Beteiligung an der Dokumentation wird diese zugänglich für die korrespondierenden Nutzergruppen.

*Die beschriebenen Regime sind in allen Variablen von den Vorstellungen der Community darüber geprägt, welches Wissen Nutzer*innen mitbringen müssen, um aus dem Wissensprodukt Nutzen ziehen zu können und darüber hinaus zum gemeinsamen Wissen beizutragen.*

Die Praktiken der Bewertung und Integration der Beiträge spielen eine zentrale Rolle bei der normativen Ausrichtung der Community. Sie sind aber strukturell gerahmt durch die formale Struktur der Rollenverteilung und der

¹⁵² Unabhängig davon gibt es immer auch Entwickler*innen, die für Unternehmen arbeiten, die bestimmte Software nutzen und daher ihre Mitarbeiter*innen anweisen können, an bestimmten Problemen zu arbeiten. Diese Zuweisungen erfolgen aber – im Gegensatz zu Canonical als Unternehmen, das explizit die Entwicklung von Ubuntu vorantreibt – außerhalb der Community-Governance.

Differenzierung von Berechtigungen und Entscheidungsrechten innerhalb der Community. Die Mitgliedschaft wird gerahmt durch formale Bewerbungsprozesse, aber auch durch die interaktive Bewertung von Beiträgen und schließlich die Rückwirkung der Wissensprodukte auf ihre Nutzer*innen. Sie bildet ein identitätsstiftendes Moment als Selbstbild der Community und es spielt insofern eine wichtige Rolle, dass in den informellen Bereichen alle Mitglieder beitragen und somit die Selektion der Mitglieder auf die Beiträge zurückwirkt.

Die Differenzierung von Expert*innen und Laien im technik-soziologischen Sinne ist vor dem Hintergrund einer auf Beiträgen basierenden Community, in der verschiedene Akteure – auch Nicht-Expert*innen – beitragen dürfen, nicht haltbar. Vielmehr differenzieren sich spezifische Beitragsmöglichkeiten für verschiedene Experten- oder Wissensstufen der Mitglieder der Communities aus. Die spezifischen Grenzziehungen prägen die Wissensproduktion auf vielfältige Weise. Beispielhaft werde ich im nächsten Abschnitt die Strukturierung eines vergleichbaren Softwareprodukts der Communities skizzieren.

10.3 Software: Das Wissensprodukt als Spiegel und Selektionsmechanismus der Community

Im Installations-Programm der verschiedenen Distributionen zeigen sich die normativen Setzungen der Communities sehr deutlich, wie in Kapitel 9 ausführlich beschrieben.

Schon bei der Gestaltung der Websites und der Hinführung zum Download-Link für die Installations-CD fallen unterschiedliche grafische Gestaltungen und deutliche Unterschiede in der Sprache auf, die gewisse Präferenzen und Prioritäten der Communities widerspiegeln und die vom User unterschiedliche Kenntnisse erwarten. Debian und Arch haben eine deutlich simpler gestrickte, eher text-orientierte Website, während Ubuntu auf ein optisch ansprechendes Design setzt. Sicher spricht die einfache Gestaltung der Debian- und Arch-Seiten auch an, aber vermutlich eben eher technisch orientierte User. Somit wird schon vor dem Start der Installationsroutine ein Selektionsprozess auf dem Weg zum Installationsmedium wirksam.

Das Installationsmedium

Auf dem Weg zum Installationsmedium legen Debian und Arch dem User einen Download über technisch raffinierte Wege nahe, die Netzwerk-Ressourcen schonen und dem Community-zentrierten Gedanken sowie dem FLOSS-Prinzip entsprechen. Die Bereitstellung großer Dateien, wie bspw. das Installationsmedium (oft über ein Gigabyte), erzeugt im Vergleich zu einfachen Websites eine große Menge an ausgetauschten Daten. Anstelle des Downloads von einer zentralen Website können die Daten daher direkt zwischen den Usern ausgetauscht werden. Dies verteilt die benötigte Last und die Verwendung der dafür beanspruchten Ressourcen auf eine große Zahl von Usern und muss nicht komplett vom Anbieter einer einzelnen Website getragen werden. Dies stellt aber spezifisches Wissen dar, das den meisten Internet-Usern heute nicht mehr geläufig ist. Daher bietet Ubuntu prominent einen Download über das gewöhnliche Website-Protokoll http an. Hier zeigt sich eine interessante Varianz zwischen niederschwelligem Zugang für Laien (Ubuntu) und technisch versierten Ansätzen der Techniknutzung.

Wird das Installationsmedium gestartet, hebt sich der Installer von Ubuntu stark von den anderen beiden ab. Auf dem Start-Bildschirm kann direkt die Sprache ausgewählt werden und der User hat die Wahl, das System zunächst als ein sogenanntes *Live-System* (ohne Veränderung der bestehenden Rechnerkonfiguration) zu starten und somit ein komplettes System inklusive Browser, Mail-Programm und Office-Paket einfach vor einer Installation auszuprobieren – ein Feature, das für Einsteiger*innen deutlich wichtiger ist als für erfahrene User.

Debian stellt einen einfachen grafischen Installationsassistenten zur Verfügung. Arch hingegen bietet keinerlei grafische Oberfläche an, sondern lediglich eine Kommandozeile, auf der der User Schritt für Schritt selbst die Installation durchführen kann. Bei Bedarf kann er allerdings auf die auf dem Medium hinterlegte Dokumentation zurückgreifen. Das Installationsmedium beschränkt sich also auf das Mindeste, nämlich die Kommandozeile, und verzichtet auf einen Automatismus. Somit wird der User dazu angehalten, sich mit dem System im Laufe der Installation auseinanderzusetzen und sich das notwendige Wissen dafür anzueignen.

Hürden bei der manuellen Konfiguration

Besonders deutlich werden die Unterschiede zwischen den Skripten bei der Umsetzung der Konfiguration der Tastatur. Das Tastaturlayout muss bei

Arch über die Eingabe von Befehlen über die Tastatur angepasst werden, was besonders schwerfällt, da man dafür ja die richtigen Tasten finden muss. Entweder weiß man, wo die Tasten bei einer englischen QWERTY-Tastatur üblicherweise liegen (die zunächst voreingestellt ist), oder man probiert alle Tasten aus oder man hat einen zweiten Rechner mit Internet-Zugang und besorgt sich dort die Informationen. „Manuell“ muss bei Arch etwas eingegeben werden wie: `loadkeys de-latin1`. Erst jetzt befinden sich hinter allen Knöpfen einer deutschen Tastatur auch die darauf abgebildeten Symbole – insbesondere Sonderzeichen wie `*/-` sind nun leicht zu finden. Dies mag für einen technisch versierten User nicht weiter dramatisch sein, weil er „vollstes Verständnis“ dafür hat, dass eben die Tastatur erst konfiguriert werden muss – woher soll der Computer sonst auch ‚wissen‘, welche Beschriftungen die Tasten haben –, aber für einen Laien stellt sich dies möglicherweise schon als unüberwindbar Hürde dar und er bricht die Installation frustriert oder entnervt ab. Bei Debian und Ubuntu kann man sich per Pfeiltasten oder gar über die Maus durch die Sprachoptionen klicken und erspart sich so das Rätselraten. Bei Ubuntu ist sogar eine automatische Tastaturerkennung möglich, bei der der User aufgefordert wird, bestimmte Tasten zu drücken, und das Programm aufgrund der Eingabe ‚errät‘, welche Tastaturbelegung die richtige ist.

Eine Kenntnis der verschiedenen Tastaturbelegungsoptionen ist hier durch die grafische und automatisierte Unterstützung des Assistenten nicht notwendig. Insbesondere bei Arch fällt auf, das man sich durch eine unglaublich lange Liste verschiedener Tastaturbelegungen in Abhängigkeit der verwendeten Rechnerarchitektur arbeiten muss, um die richtige zu finden (Abb. 9.20). Positiv gewendet lernt der Arch User bei der Auseinandersetzung mit dem Tastaturlayout, sich auf der Kommandozeile zu bewegen, Ordner zu durchsuchen, ist angehalten, sich mit der Rechnerarchitektur seines Computers zu beschäftigen und weiß künftig den Befehl für die Auswahl der Tastaturlayouts. Dies erfordert jedoch eine gewisse Bereitschaft, Zeit und Energie aufzubringen und sich mit der Technik auseinanderzusetzen. Somit scheiden hier einige potenzielle User wohl aus. Andere User, die sich erfolgreich durch die Befehle arbeiten, empfinden möglicherweise eine gewisse Befriedigung durch den erreichten Erfolg.

Auch hier spiegeln sich also die Präferenzen der Community wider – einfache Benutzerführung vs. direkte Konfigurierbarkeit – und durch das Software-Skript werden jeweils bestimmte Nutzertypen selektiert.

Vorausgesetzte Fachbegriffe

Vom User werden im Laufe des Installationsprozesses unterschiedliche Kenntnisse und Fähigkeiten verlangt und auf unterschiedliche Weise muss der User sich mit dem Rechner, dem Betriebssystem auseinandersetzen. Besonders deutlich wird dies bei der Vorbereitung der Festplatte für die Installation. Sowohl Ubuntu als auch Debian schlagen eine Partitionierung der Festplatte vor und beschreiben verschiedene Möglichkeiten. Bei Ubuntu erfährt der User bei der Option „Use LVM“ beispielsweise, dass der „Logical Volume Manager“ es ermöglicht, zu einem späteren Zeitpunkt die Größe der Partitionen zu ändern. Bei Debian wird davon ausgegangen, dass der User weiß, was *LVM* oder auch *encrypted LVM* bedeutet – oder zumindest nach der Vorkonfiguration anhand der angezeigten detaillierten Partitionierung dann versteht, was der Installer im nächsten Schritt durchführen wird. Der Debian-Installer bietet dann auch die Möglichkeit an, die Vorkonfiguration noch in einem weiteren Schritt manuell anzupassen. Diese Option gibt es bei Ubuntu nicht – hier wird nach einer Bestätigung die vorgeschlagene Konfiguration durchgeführt und sofort mit dem Kopieren der Dateien begonnen, während der User die Einstellungen zu Lokalisierung, User-Account und Rechnername vornimmt.

Gruppierung von Konfigurationsoptionen

Beim Rechnernamen zeigt sich ein weiterer Unterschied der Installer. Bei Ubuntu befindet sich eine Eingabemaske für Name des Users, Passwort und einem weiteren Feld für den Rechnernamen. Die Einstellung erscheint hier also im Kontext mit einer Reihe anderer Namensgebungen (Kontoname, Benutzername) und ist basierend auf der Typbezeichnung des Rechners vorausgefüllt. Der Kommentar zum Rechnernamen ist lediglich: „Der Name, der bei der Kommunikation mit anderen Rechnern verwendet wird.“ Im Heimnetzwerk spielt dies eine untergeordnete Rolle und der User wird hier nicht mit weiteren Details belastet. Bei Debian erfolgt die Benennung des Rechners im Zusammenhang mit der Netzwerkkonfiguration und die Beschreibung der Funktion des Rechnernamens ist deutlich ausführlicher. Dadurch erscheint die Einstellung in einem technischeren Kontext und ist für unbedarfte User schwieriger einzuordnen. Bei Arch erfolgt diese Einstellung durch die Bearbeitung einer Textdatei, ist also auch hier wieder manuell zu konfigurieren.

Benutzerführung und Skripting der User

In Debian arbeitet sich der User durch mehrere Bildschirme, wird sukzessive mit neuen Informationen versorgt und soll darauf basierend die für ihn passenden Entscheidungen treffen. Dabei ist für einen Laien ungewiss, welche weiteren Entscheidungen noch getroffen werden müssen, und erst als letzter Schritt wird die Festplatte konfiguriert und anschließend der eigentliche Installationsprozess, das Kopieren der Dateien, gestartet. Zu diesem Zeitpunkt mag mancher in Zweifel geraten sein, ob alle Informationen richtig interpretiert wurden und die richtigen Einstellungen getroffen sind. Ganz gegensätzlich dazu wird bei Ubuntu zuallererst die Grundsatzentscheidung der Festplatten-Einteilung getroffen, und dann, während der Kopiervorgang der Installation nebenher läuft, werden die Informationen für die Konfiguration abgefragt.

Arch fällt hier völlig aus dem Vergleich, weil der User Schritt für Schritt alles selbst macht. Er partitioniert die Festplatte selbst, und zwar mithilfe eines Kommandozeilenprogramms (vgl. Abb. 9.21, S. 242), über die Eingabe der Festplattensektoren. Er bindet die Festplatte selbst ein und kopiert die Systemdateien darauf. Der Rechnername wird dabei über die Erstellung einer Datei `/etc/hostname` definiert. Dazu muss der User auf der Kommandozeile einen Texteditor aufrufen und die Eingabe an der richtigen Stelle speichern. Durch die *manuelle* Konfiguration des Systems lernt der User – wenn er es nicht schon kann – die wesentlichen Elemente, Konfigurationsdateien und die wichtigsten Befehle für die Verwendung der Kommandozeile und die entsprechenden Programme kennen. Dies ist explizit gewollt, da so der User befähigt wird, künftig seine Probleme selbst zu lösen, da er notwendigerweise im Zuge der Installation ein Grundverständnis des Systems erlangt.

Ein weniger erfahrener User kommt bei Arch also wohl nicht über den Startbildschirm hinaus und wendet sich einem anderen System zu. Das ist durchaus gewollt. Die Installationsskripte von Debian und Ubuntu bieten hier klare Strukturen, an denen sich die User orientieren können. Ubuntu verbirgt schließlich soweit möglich die Komplexität des Systems, um auch einfache Nutzer*innen zum Ergebnis eines installierten Systems zu führen. Mit anderen Worten erfährt der einfache User bei Ubuntu eine starke Führung durch das Skripting der Software. Bei Debian bestehen mehr Wahlmöglichkeiten, die aber auch eine gewisse Stufe von Wissen erfordern. Arch bietet schließlich die maximale Wahl-Freiheit, man muss sich aber dafür intensiv das notwendige Wissen erarbeiten.

Verschiedene Grade von Wissen werden erwartet

Für eine erfolgreiche Installation muss ein Arch User also notwendigerweise Dokumentationen lesen (RTFM) und benötigt somit allermindestens *Primary Source Knowledge*, eignet sich aber im Laufe der Installation ein Grundwissen an, das schon in Richtung *Interactional Expertise* geht, da Zusammenhänge der Funktionsweise des Systems zutage treten, die schließlich auch in der Interaktion mit Entwickler*innen und anderen erfahrenen Usern hilfreich sind.

Bei Debian kommt man grundsätzlich mit einem weitläufigeren Grundverständnis des Systems aus, das man sich auch aus Computer-Zeitschriften aneignen kann. LVM, Partitionierung und dergleichen werden dort diskutiert und können auch ohne die Einarbeitung in die Befehle und Dokumentationen über den Installer erreicht werden – daher reicht hier ein *Popular Understanding* von Computern aus.

Bei Ubuntu ist lediglich schematisches Wissen ausreichend. Dem User werden einfache Hinweise gegeben, was die Optionen bedeuten: „LVM allows easier partition resizing.“ Dies ist schon sehr nahe dran am Bier-Deckel-Wissen von Collins und Evans (2007) – *Beer-Mat Knowledge* ist ausreichend für die Installation.

Der Installer als Spiegel der Konfiguration der epistemischen Regime

Die Anforderungen für die erfolgreiche Installation stellen die Mindestvoraussetzungen für eine passive Teilnahme an der Community dar. Außerdem spiegelt sich darin das Nutzerbild wieder: Der „Telly User“ kann sich mit Ubuntu über wenige Klicks ein fertiges Betriebssystem zaubern, der „Power-User“ erhält in Debian Informationen, auf deren Grundlage er selbst Entscheidungen für die Konfiguration des Systems treffen darf, der „Linux-Lerner“ erhält mit Arch zunächst eine Kommandozeile, in der er sich ausprobieren darf und Schritt für Schritt über die Auseinandersetzung mit der Dokumentation befähigt wird, ein Linux manuell zu installieren.

Dementsprechend sind auch die Anforderungen an potenziell Beitragende gesteckt: Der Ubuntu-User darf gerne Rückmeldung über seine *User Experience* geben, etwaige Schwierigkeiten oder Unverständlichkeiten bei der Menüführung, oder gar „Eselsohren“ im Design beheben (sogenannte *Papercut Bugs*, s. S. 192). Für Debian ist für das Beitragen ein gewisses technisches Verständnis hilfreich, um beispielsweise die Mailingliste oder das Bug-Reporting-Tool richtig zu benutzen. Der Installer impliziert durch Design und

Gestaltung eine eher technisch-rationale Attitude, die bei der Interaktion mit der Community ebenfalls hilfreich ist. Arch-User haben eine relativ hohe Einstiegshürde für die erfolgreiche Installation, qualifizieren sich aber damit auch für die weitere Aneignung von Expertise, die für sinnvolle Beiträge für die Community auch gefordert wird.

Die Strukturen und Verfahren spielen hier eine nachrangige Rolle, da sie vermitteln zwischen dem Selbstbild der Community und der Gestaltung der Beiträge. Anhand des Vergleichs der Variablen Beiträge und Mitgliedschaft wird aber anschaulich, dass sich grundlegende Werte der Community von der Mitgliedschaft über die Akzeptanz der Beiträge bis hin in die Gestaltung der Software ziehen. Der Installer ist also bei Ubuntu die in Software implementierte Inklusion von Laien sowie bei Arch die Exklusion der Laien oder die eingeschriebene Lernfunktion auf dem Weg zu einem qualifizierten Arch-User.

Die spezifische normative Konfiguration des epistemischen Regimes, die strukturierenden Werte der Community, fließen also in das gemeinsame Wissensprodukt und erfüllen somit auch eine Selektionsleistung der potenziellen Mitglieder der Community und informieren die User der Software.

Das gemeinsam produzierte Wissen wirkt als Träger der Werte der Community und wirkt über den Rückgriff der User auf die Dokumentationen und die Benutzung der Software auf die Mitglieder und die Verfassung der Community zurück. Das hervorgebrachte Wissensprodukt trägt somit einen essenziellen Teil zur Strukturierung des epistemischen Regimes bei.

11 Epistemische Regime verschiedener Experten-Laien-Differenzen

Es wurde deutlich, wie in den betrachteten FLOSS-Communities verschiedene Variablen zusammenwirken und epistemische Regime bilden, die unterschiedliche normative Vorstellungen über die Gestaltung von Software fest-schreiben und sich dabei grundlegend darin unterscheiden, wo die Grenze gezogen wird zwischen legitimen und illegitimen Beiträgen. Auf diese Weise wird das gemeinsame Wissen geformt, das wiederum zurückwirkt auf die Mitglieder der Community – einerseits über die Inhalte und die Strukturen des Wissens und andererseits über die Inskriptionen der produzierten Software.

Darüber hinaus ergeben sich weitere Erkenntnisse über die Wirkungsweise der sozialen Organisation, der Normen und Praktiken der Communities sowie die Ausgestaltung einer strukturellen Differenz zwischen Hersteller*innen und Nutzer*innen und zwischen Expert*innen und Laien, die durch epistemische Regime organisiert und konfiguriert wird.

11.1 Epistemische Regime als stabile normative Ordnungen

Für den analytischen Vergleich der untersuchten Communities habe ich das Konzept epistemischer Regime aufgegriffen und für meine Analyse operationalisiert. Dafür habe ich drei elementare Variablen epistemischer Regime formuliert, die es ermöglichen, die epistemischen Regime in ihrer Funktion und Wirkung zu erfassen.

Grundlage des Vergleichs waren drei unterschiedliche FLOSS-Communities, die dasselbe Ziel anstreben, ein performantes Wissensprodukt zu generieren, das Computer durch ein gemeinsam produziertes Betriebssystem als Technik nutzbar macht. Für die Entwicklung der Software und der damit verbundenen Dokumentation der Funktionsweise und den Hilfestellungen werden dabei in allen Fällen die in FLOSS-Projekten üblichen Praktiken und

Methoden verwendet. Software wird kollaborativ entwickelt, zunächst von einer Gruppe von Core-Developern in gegenseitiger Absprache. Verteilte Akteure können Code-Beiträge anbieten, die von Core-Developern integriert werden. Für die Koordination der Aufgaben gibt es Feedback-Systeme, in denen Fehler gemeldet und Wünsche über neue Funktionen geäußert werden können. Darüber hinaus gibt es Dokumentationen über die Konfiguration und Funktionsweise der Software, die üblicherweise auf kollaborativen Websites angefertigt werden, sogenannten Wikis, und kommunikative Interaktionen über konkrete Problemfälle, die über Mailinglisten oder Diskussionsforen archiviert werden und somit Hilfestellung bieten. Dokumentation und Support werden interaktiv generiert, hier gibt es keine formale Hürde für die Teilnahme, die Qualitätskontrolle ergibt sich aus der Diskussion einer Vielzahl von Teilnehmer*innen.

Jedoch zeigen sich bei genauerer Betrachtung sehr starke Unterschiede in der sozialen Organisation der Communities sowie unterschiedliche Gestaltungen des produzierten Wissens. Obwohl sich alle Fälle als Meritokratien verstehen, gibt es doch sehr deutliche Unterschiede darin, welche Beiträge als Verdienst erachtet werden und auf welche Weise Rechte verteilt werden; den Beitragenden wird ermöglicht, direkt gestalterisch auf das gemeinsame Wissen zuzugreifen.

Hier zeigt sich, dass sich trotz aller struktureller Parallelen und Gemeinsamkeiten der FLOSS-Produktion in den spezifischen Communities starke Differenzen ergeben, die offensichtlich zu unterschiedlichen Ergebnissen in der Wissensproduktion führen.

Zwar sind sich die Wissensprodukte wiederum ähnlich in der Hinsicht, dass sie als Betriebssystem der Hardware Leben einhauchen, über unzählige Applikationen Funktionen bereitstellen und für die sinnvolle Verwendung der Software entsprechende Dokumentation und Hilfestellung anbieten. Jedoch unterscheiden diese sich sehr stark in der Zugänglichkeit für verschiedene Nutzergruppen, die unterschiedliche Kenntnisse über die Funktionsweise der Software haben. Entsprechend sind sowohl Wissen als auch Software in ihrer Gestaltung auf unterschiedliche Bedürfnisse zugeschnitten.

Wie in Kapitel 6 ausgeführt, beschreibt der Begriff epistemischer Regime die normative Strukturierung von Wissen. In den beschriebenen Fällen konnte ich nun herausarbeiten, wie sich verschiedene normative Vorstellungen zeigen darüber, was als legitimer Beitrag gilt und auf welcher Grundlage Mitglieder der Community Rollen erhalten können, die Einfluss auf die Gestaltung ermöglichen. Dabei zeigt sich, wie normative Grundeinstellungen

die Regime durchziehen. Die zugrunde liegenden Normen stabilisieren die Regime durch eine orientierende Struktur und prägen das Wissensprodukt. Kriterien der Gültigkeit von Beiträgen sind eben nicht universell, sondern auch eine Frage normativer Vorstellungen. Problemlösungen durch Algorithmen lassen sich auf verschiedene Weise implementieren und schließlich lässt auch die Integration von vorhandenen Lösungen aus dem Upstream große Spielräume offen, die verhandelbar sind.

Die geteilten Normen erfüllen dabei in den Communities, die aus freiwilligen Beitragenden bestehen, eine Identifikationsfunktion, die zugleich den Beitrag für die Community motiviert, und überdies eine Selektionsfunktion, die die Mitglieder-Struktur der Community kontrolliert und damit auch die informell organisierten Beiträge strukturiert. FLOSS-Communities bilden für die Untersuchung der Strukturen Wissen produzierender Gemeinschaften einen idealen Analysegegenstand, da sich die Communities relativ gut über die Kommunikationskanäle der Communities eingrenzen lassen, es einen zentralen Wissenspool gibt und eine zentrale Organisation, die Strukturen, Rollen und Beitragsmodalitäten definiert.

Die normative Grundierung der betrachteten Fälle hat ihren Ursprung dabei maßgeblich in der Motivationen der Gründer der Projekte, Ian Murdock (Debian), Aaron Griffin (Arch) und Mark Shuttleworth (Ubuntu). Diese hatten von Anfang an klare Ziele, die sie mit ihrer Distribution erreichen wollten: eine transparent-bürokratische, technisch elaborierte Community-Distribution, eine „technisch einfache“¹⁵³ Distribution für eine technische Avantgarde bzw. ein Linux für Laien. Dieser „Geist“ der Gründerväter prägte die Verfassungen der Communities und zog andere Entwickler*innen und User an, die sich mit diesem Geist identifizieren konnten und fand so eine kritische Masse an Individuen, die über die formale und informelle Strukturierung der Community und über geteilte Kriterien bei der Gestaltung der Beiträge ein stabiles epistemisches Regime begründen. Die ursprüngliche Konfiguration des Gründers und der ersten Mitglieder schreibt formale Strukturen fest und etabliert bestimmte Normen in Formen von Anforderungen und Kriterien an die gemeinsamen Produkte, aber auch an die neuen Mitglieder, die zur Community dazustoßen. Im spezifischen Fall von Open-Source-Software wirkt das performante Wissensprodukt als sichtbarer und erfahrbare Träger der epistemischen Normen der Community und wirkt

153 im Sinne der UNIX- und Hacker-Philosophie „Keep it simple, stupid“

seinerseits als Verbreitungsinstrument gleichsam einer Werbepattform und gleichzeitig als Selektor, der User mit Präferenzen anspricht und erreicht, die die Werte der Gemeinschaft teilen, andere aber eher abwehrt.

Dabei verstehen sich bemerkenswerterweise alle betrachteten Fälle explizit als Meritokratien, unterscheiden sich aber aufgrund ihrer normativen Struktur deutlich darin, welche Art von Beiträgen als verdienstvoll angesehen wird und mit Anerkennung im Sinne von partizipativen Rechten verbunden ist. Jenseits einer mit Einfluss verbundenen Anerkennung gibt es aber auch symbolische Formen der Anerkennung, die die Identität und Zugehörigkeit der Mitglieder unterstützen und motivierend wirken. So kann jeder Ubuntu-User über die Signierung des *Code of Conduct* auf der gemeinsamen Kollaborationsplattform *Launchpad* als *Ubuntero* sichtbar werden. Und offiziell anerkannte *Ubuntu Members* bekommen sogar ein schriftliches Zertifikat. Die Bewertung und Anerkennung von Beiträgen als Leistung ist das Kernelement der Wissensproduktion, da durch die Beiträge das Wissen strukturiert wird und die Werte der Community in das Wissensprodukt eingeschrieben werden. Das Beitragen wird reguliert durch formale Rollen, die Qualifikationsverfahren erfordern und mit Zugriffsrechten versehen sind, sowie Entscheidungsstrukturen, die übergreifende Zusammenhänge regulieren und richtungsweisende Entscheidungen treffen. Eine informelle Regulierung des Beitragens geschieht aber ebenfalls durch die formalen Mitglieder, da sie in den Bereichen, die keine formalen Zugriffsbeschränkungen haben, dennoch durch ihre Privilegien der Selektion und Integration der Beiträge die Qualität interaktiv kontrollieren. Ferner haben die formalen Mitglieder über ihre Stimmrechte Einfluss auf die formalen Strukturen der Community und können – je nach Verfassung des epistemischen Regimes – diese Strukturen verändern.

Durch die beschriebenen Einflussmöglichkeiten auf die Gestaltung der Beiträge und die formalen Strukturen bildet die formale Mitgliedschaft eine Schlüsselvariable der epistemischen Regime. Darüber hinaus verkörpert die formale Mitgliedschaft durch die fest formulierten Anforderungen an Bewerber*innen auch einen wesentlichen Teil des Selbstbildes der Community. Dieses Selbstbild hat einen mittelbaren Einfluss auf das Wissensprodukt, das Wissensprodukt wirkt zurück auf die Zusammensetzung der Mitglieder. Dieses Wechselverhältnis stabilisiert das epistemische Regime.

In Abschnitt 10.1 habe ich die historischen Entstehungsmomente der epistemischen Regime skizziert. Die Unterscheidungen in der Grenzziehung zwischen verschiedenen Beitragenden bei ihrer Integration in die formalen

Strukturen sind also integraler Bestandteil der Gründungsmotivation der betrachteten Communities. Darin spiegelt sich auch eine historische Veränderung der Grenzziehung von Expert*innen und Laien wider (vgl. Kap. 1). Software weitet sich aus und wird von einer Technik, von und für Expert*innen, zu einer Technik, die auch weniger technische, alltägliche Anwendungsbereiche findet. Somit erscheinen auch Laien als User und ihre Bedeutung nimmt zu.

Vor dem Hintergrund eines elitären Regimes der Core-Developer in den dominanten FLOSS-Systemen der ersten Jahre¹⁵⁴ stellt Debian die Ausweitung der formalen Mitgliedschaft und Mitbestimmung auf alle Code-beitragenden Mitglieder dar. Zum Zeitpunkt der Gründung von Debian hatte der „Personal Computer“ zwar schon seine Ausbreitung in weniger technische Kreise gefunden, Linux blieb aber noch stärker im technischen Umfeld – unter Expert*innen – verbreitet. Ubuntu erweiterte die Ausweitung der Mitgliedschaft auf jegliche Beiträge, in einer Zeit mit dem expliziten Anspruch, Linux auch Laien nahezubringen. Arch schließlich steht für eine Rückbesinnung auf eine starke Begrenzung der Core-Developer auf eine schlagkräftige Elite.

Die einzelnen epistemischen Regime sind also recht stabil, aber über die Zeit lässt sich ein Wandel in der Konfiguration verschiedener Regime beobachten. Drei verschiedene Typen von Regimen und ihre unterschiedlichen Ausgestaltungen der Grenzziehung zwischen Expert*innen und Laien sind Gegenstand des folgenden Abschnitts.

11.2 Die epistemischen Rekonfigurationen der Experten-Laien-Differenz

In der Betrachtung von Software als Technik im Sinne einer für Laien benutzbaren Ressource spielt das Betriebssystem eine zentrale Rolle, da es im Wesentlichen der Nutzbarmachung des Computers dient und idealiter möglichst im Hintergrund arbeitet. Dabei kann aber die Konfiguration des Systems mit der Integration unterschiedlicher Peripheriegeräte durchaus kom-

154 Im in Abschnitt 5.2 genannten Beispiel handelt es sich um BSD, was ebenso wie Linux von UNIX abgeleitet ist.

plex werden. Darüber hinaus spielt im Zusammenhang mit der Systemsicherheit – gerade vor dem Hintergrund wachsender Sicherheitsprobleme und möglicher Angriffe aus dem Internet – die Konfiguration des Systems eine virulente Rolle, die teils durch Expert*innen vorgegeben und teils durch die User selbst vorgenommen wird. Das Verständnis der Zusammenhänge, die für die Konfiguration der Technik eine Rolle spielen, und das Vertrauen in die Hersteller*innen der genutzten Technik werden dabei zu kritischen Elementen.

Wie ich eingangs dargelegt habe (Kap. 1), ist ein Kernelement der FLOSS-Bewegung die Problematisierung von Vertrauen in eine opake Technikstruktur und einem damit einhergehenden Legitimationsproblem der Dichotomie zwischen technik-gestaltenden Expert*innen und technik-nutzenden Laien. Nicht zuletzt durch die alltägliche Strukturierung der Handlungspraktiken im Zusammenhang mit einer wachsenden Digitalisierung spielen die Pfadabhängigkeiten der Techniknutzung und die Möglichkeiten, die genutzte Technik zu gestalten oder deren Gestaltung zu beeinflussen, eine zentrale Rolle.

In den betrachteten FLOSS-Communities differenzieren sich die Mitglieder der Community in User mit verschiedenen Graden von Expertise. Ausgehend von einer Definition von Expertise, die die Kenntnis der Funktionsweise der Technik impliziert, gibt es zwar weiterhin eine Entwickler-Elite, die die Schreibrechte an den Kernkomponenten der Software hat, aber die Legitimation dieser Elite ist in Bezug auf die formalen Strukturen zu einem Teil transparent definiert und das funktionale Wissen über die Software selbst ist prinzipiell zugänglich und steht einer eigenen unabhängigen Weiterentwicklung im Rahmen einer eigenständigen Abspaltung vom Ursprungsprojekt („Fork“) zur Verfügung. Eine derartige Abspaltung erlaubt eine völlig eigenständige Verwaltung des Softwarecodes, allerdings gehen die Änderungen, die in das Ursprungsprojekt fließen, verloren oder müssen in die Abspaltung eingepflegt werden. Diese wird also generell vermieden, da sich dadurch Entwicklerressourcen aufspalten in Teilprojekte oder eben ein zusätzlicher Aufwand der gegenseitigen Synchronisation von Verbesserungen entsteht. Auch ist die Verwaltung mit administrativem Aufwand verbunden und darüber hinaus nur für Akteure mit Entwicklerfähigkeiten oder -ressourcen eine Option. Die Governance der Community eines FLOSS-Projekts ist also entscheidend im Hinblick auf die Beteiligungsmöglichkeiten verschiedener Akteure, die beitragen möchten.

Anders als üblicherweise in Sachtechnik zu beobachten, sind aber generell nicht mehr allein die „Expert*innen“ gleichbedeutend mit den „Entwick-

11.2 Die epistemischen Rekonfigurationen der Experten-Laien-Differenz 287

ler*innen“ die alleinigen „Hüter*innen der Regeln“, die die Funktionsweise der Technik definieren und zugrunde legen (vgl. Kap. 4). Vielmehr ermöglicht der allgemeine Zugriff auf die Funktionsregeln, d.h. der Softwarecode, die Aneignung von Wissen für einen erweiterten Kreis von Akteuren, der eben durch das epistemische Regime der Community bestimmt wird. Es entstehen hierbei verschiedene Ausprägungen der Grenzziehung zwischen Akteuren, die mitgestalten dürfen, und Akteuren, die nicht mitgestalten dürfen – vermittelt über die Selektion von Beiträgen. Die Grenzziehung beruht auf einer normativen Grundannahme, wer seine Perspektive in die Gestaltung des Wissensprodukts einbringen soll, und korrespondiert mit unterschiedlichen Vorstellungen, welcher Grad von Expertise, welche Arten von Wissen hilfreich und sinnvoll für die Wissensproduktion sind.

Es bleibt also zu unterscheiden zwischen der Expertise oder dem Wissenslevel verschiedener Akteure auf der einen und dem Zugriff auf die „Regeln“ der Technik, beziehungsweise den Einfluss auf die Gestaltung des gemeinsamen Wissensprodukts, auf der anderen Seite. Dafür habe ich in Kapitel 5.1 Expertise in verschiedene Stufen unterteilt und empirisch untersucht, welche Akteure verschiedener Wissensstufen in welchem Maße in verschiedenen FLOSS-Communities eingebunden sind.

Die Neuordnung des Verhältnisses von Expertise und Macht über die Gestaltung der Regeln der Technik in den untersuchten Communities werde ich in den folgenden Abschnitten als Konfigurationen epistemischer Regime diskutieren.

11.2.1 Differenzierungen zwischen den Entwickler*innen

In Debian gibt es ein sehr aufwendiges, aber stark formalisiertes Bewerbungsverfahren zur Erlangung des Status eines *Debian Developers*. Die *Debian Developer* sind zunächst alle gleichberechtigt, haben aber spezifische Zuständigkeiten für bestimmte Bereiche des Systems.

Bei Ubuntu und Arch gibt es hingegen eine Unterscheidung eines Haupt-Bereiches („Main“ bzw. „Core“ und „Extra“) und eines Community-Repositories („Universe“ bzw. „Community“).¹⁵⁵ Zudem gibt es sowohl in Arch als auch in Ubuntu einen Rollentyp, der nicht über ein formales Verfahren

155 Die Repositories, die sich in restriktivere Lizenzen gruppieren („Multiverse“ und „Restricted“ in Ubuntu bzw. ‚Contrib‘ und ‚non-free‘ in Debian) spare ich an dieser Stelle aus, da sie sich nicht auf die Relevanz, sondern auf die Lizenzierung beziehen.

reguliert ist. *Arch Developer* – die Gesamtheit der Core-Developer – rekrutieren neue *Arch Developer* in eigener Regie und in Ubuntu gibt es einen Kreis von Entwickler*innen, die durch das Unternehmen Canonical angestellt werden.

Nichtsdestotrotz sind deren Aktivitäten öffentlich einsehbar über die Kollaborationssysteme und die Ubuntu-Angestellten sind schließlich eingebunden in die Community-Strukturen und somit auch an die Entscheidungen der Gremien der Community gebunden, in denen aber auch der Gründer und Sponsor von Canonical eine entscheidende Rolle innehat. Außerdem gibt es sowohl in Arch als auch in Ubuntu formale Verfahren für die Erlangung eines Entwickler-Status. Im Gegensatz zu Ubuntu kann bei Arch über ein formales Verfahren aber nur eine geringer privilegierte Entwickler-Rolle erreicht werden. Diese Rolle wird bezeichnenderweise *Trusted User* genannt – bezeichnend, weil hier offensichtlich Entwickler-Tätigkeiten auch in den Bereich der User fallen und gleichzeitig eine Relevanzunterscheidung zwischen den „echten“ Entwickler*innen, die die zentralen Komponenten betreuen, und den Usern, die erweiterte Software verwalten, gesetzt wird. Bei Debian gibt es neben der privilegierten Entwickler-Rolle (*Debian Developer*) auch eine weniger privilegierte Rolle (*Debian Maintainer*), diese wird aber nur als Übergangstatus zum bzw. zur voll privilegierten Entwickler*in betrachtet. In Debian und Ubuntu kann also jede*r über ein formales Verfahren bei Erfüllung der notwendigen Kriterien in den Kreis der Entwickler*innen aufgenommen werden.

Über die Anträge wird in einem gesonderten Gremium entschieden, dessen Besetzung von der Entwickler-Community mitbesetzt wird. Bei Arch bleibt der Kreis der Entwickler*innen eine Elite, die selbst neue Mitglieder in ihren Kreis erwählt. In Tabelle 11.1 werden die verschiedenen Rollengestaltungen gegenübergestellt.

Tab. 11.1: Vergleich der Entwicklerrollen (Auswahl)

Zugriffsbereich	Ubuntu	Debian	Arch
Core Repositories	Core-Developer	<i>Debian Developer</i>	<i>Arch Developer</i>
erweiterte Repositories	<i>Masters of the Universe</i>	<i>Debian Developer</i>	<i>Trusted User</i>
paketbasierter Zugriff	<i>Contributing Developer</i>	<i>Debian Maintainer</i>	–
individuelle Pakete	User	–	User

11.2 Die epistemischen Rekonfigurationen der Experten-Laien-Differenz 289

Darüber hinaus kann FLOSS-typisch jeder User individuelle Änderungen für sich selbst vornehmen und in Arch und Ubuntu gibt es auch eine Möglichkeit für User, ihre persönlichen Softwarepakete anderen Usern bereitzustellen. Die Möglichkeit individueller Modifikationen der Software birgt immer auch die Option, dass sich ein großer Teil von Mitgliedern abspaltet und gemeinsam eine Abspaltung als eigenes Produkt weiterentwickelt. Die Hoheit über Kernelemente der Software ist also relativ, da die (Core-)Entwickler*innen ein gewisses Interesse haben, im Sinne der Community zu entscheiden, um eine Spaltung zu vermeiden.

Beteiligungsmöglichkeiten an der Entwicklung

Bei Debian gibt es also eine sehr hohe Schwelle, um in den Kreis der Entwickler*innen aufgenommen zu werden; aber sobald man den Status erreicht hat, genießt man auch weitreichende Mitbestimmungsrechte. Die Rolle ist aber fokussiert auf Beitragende, die auf Code-Ebene beitragen, also im hier verwendeten Sinne der *Contributory Expertise* (vgl. auch nächster Abschnitt). Es gibt also eine breite Schicht von Entwickler*innen, die gemeinsam das Projekt steuern.

Bei Ubuntu existiert insofern eine niedrigere Schwelle, um Entwickler*innen zu werden, da es offenbar keine derart formalisierte Überprüfung des Fähigkeiten und Intentionen wie bei Debian gibt, sondern Anwärter*innen hier mehr informell über die Kontribution und Interaktion mit den Entwickler*innen hineinwachsen. Außerdem gibt es die Möglichkeit, unabhängig vom Beitrag zur technischen Entwicklung, eine formale Mitgliedschaft zu erlangen, allerdings ohne Schreibrechte am Repository. Durch die Ausdifferenzierung der Rollen ist bei Ubuntu also generell die Erlangung einer Mitgliedschaft leichter; Schreibrechte für zentrale Teile des Systems (Core) zu bekommen, gestaltet sich allerdings schwieriger. In dieser Hinsicht erscheint Debian als egalitärer, insofern man die hohe Schwelle erreicht hat.

Bei Arch ist von vornherein kein Verfahren vorgesehen, um Schreibrechte für die zentralen Teile des Systems zu bekommen. Lediglich für das *Community-Repository*, also die Software-Pakete, die jenseits der wichtigen Funktionalitäten liegen, kann man als Trusted User Schreibrechte bekommen.

Ungeachtet der Entwicklerrollen kann jede*r allgemein auch ohne formale Rolle Code-Änderungen anbieten, aber ob diese dann integriert werden, hängt eben von den zuständigen formalen Entwickler*innen des Projekts ab. Jenseits der Entwicklung am Softwarecode gibt es auch viele andere Beteiligungsmöglichkeiten, sie sind aber je Community mit unterschiedlichen An-

forderungen verbunden und korrespondieren auf verschiedene Weise mit formaler Anerkennung durch eine Rolle in der formalen Organisation.

11.2.2 Beiträge jenseits einer „Contributory Expertise“

In meiner Rekonzeptionalisierung der Expertise-Begriffe von Collins und Evans (2007) bezieht sich *Contributory Expertise* auf die Beteiligung an der Gestaltung der *Regeln der Technik*, also der *Veränderung* des Softwarecodes. Demzufolge beschreibt *Interactional Expertise* die Kenntnis der Funktionsweise der Software oder die Fähigkeit, mit den Entwickler*innen auf Augenhöhe zu kommunizieren, ohne selbst Softwarecode beizutragen. Als Beitragende verstehe ich dabei explizit auch jene, die keine *Contributory Expertise* im technischen Sinne haben, sondern in anderen Bereichen als der Code-Entwicklung beitragen. Der Begriff der *Interactional Expertise* spielt folglich eine Rolle bei den informell organisierten Beitragsformen des Feedbacks über Fehler und Änderungswünsche und beim Beitragen zum gemeinsamen Wissenspool. Hierbei handelt es sich also um Beiträge, die nicht direkt die Funktionsweise der Software verändern, sondern Fehlfunktionen aufzeigen oder gewünschte Veränderungen formulieren sowie Beiträge, die die Funktionsweise der Software vermitteln und deren Konfiguration und Anwendung beschreiben. Hierfür ist es sinnvoll, die Funktionsweise der Software zu kennen und die Sprache der Entwickler*innen zu beherrschen, denn je präziser Fehlfunktionen beschrieben und gewünschte Funktionen begründet werden können, desto größer ist die Anschlussfähigkeit des Feedbacks für die Entwickler*innen und eine Umsetzung wahrscheinlich.

In den FLOSS-Communities existieren also jenseits der streng formal regulierten Software Repositories Bereiche, die informell reguliert sind und in denen User beitragen können, die selbst nicht aktiv entwickeln, sich aber hinreichend mit der Funktionsweise der Software auseinandersetzen, um im Hinblick auf Fehlfunktionen oder fehlende Funktionen hier in Interaktion mit den Entwickler*innen zu treten. Insbesondere in den Beitragsformen, die näher an der technischen Entwicklung des Softwarecodes sind, ist eine stärkere *Interactional Expertise* sinnvoll – und je nach Community auch Voraussetzung. Bemerkenswerterweise gibt es hier in Ubuntu eine stärkere Akzeptanz und Integration von Beiträgen, indem fehlerhafte Fehlerberichte „triagiert“ und in eine für Entwickler*innen anschlussfähige Form gebracht werden können. Auch im Bereich der Dokumentation der Software ergibt

11.2 Die epistemischen Rekonfigurationen der Experten-Laien-Differenz 291

sich die Möglichkeit einer stufenweisen Vermittlung zwischen dem technischen Bereich der Software und der Anwendungsperspektive auf Funktion und Konfiguration, die unterschiedlich tiefgehende *Interactional Expertise* benötigen. Sowohl im Bereich des Feedbacks als auch im Bereich der Dokumentation sind auch Beiträge, die unter der Schwelle von Expertise liegen, möglich und je nach Konfiguration des epistemischen Regimes auch akzeptiert und valide.

Generell werden Beiträge in den informellen Bereichen Feedback, Dokumentation und Support interaktiv entwickelt und Teilnehmer*innen können die Fragen, Anleitungen und Fehlerberichte Anderer durch ihr eigenes Zutun verbessern. Allerdings unterscheiden sich die hier betrachteten Fälle deutlich in der Akzeptanz verschiedener Beiträge. Das Triagieren von Fehlerberichten als explizite Beitragsform in Ubuntu ist eine Folge einer hohen Akzeptanz von Beiträgen, die von weniger erfahrenen Usern erbracht werden und daher in ihrer Ausgangsform nicht anschlussfähig für Entwickler*innen sind. Üblicherweise werden die Beitragenden gefragt, ob sie mehr Informationen zum auftretenden Fehler beisteuern können – wenn nicht, wird der Bericht ignoriert. Hier setzt die Notwendigkeit von *Interactional Expertise* ein, denn die Nachfragen müssen verstanden werden und um die passenden Antworten zu geben, ist Kontextwissen erforderlich. Je größer die Varianz an Expertise oder Wissenslevel an dieser Stelle ist, desto eher können verschiedene User hier übersetzen und vermitteln. Die Bereitschaft und Geduld, hier zu vermitteln, ist sehr unterschiedlich. Die Varianz an Expertise sowie die Bereitschaft zur Vermittlung sind dabei innerhalb der Community, aber auch individuell verschieden. In jeder Community kann ein Beitrag zurückgewiesen werden, aber im Kontext betrachtet ergeben sich Community-spezifische Tendenzen. Diese Tendenzen formen epistemische Regime, die die Validität verschiedener Beiträge und den Umgang mit Beiträgen verschiedener Qualität implizit regulieren.

Da diese Bereiche des Beitragens nicht (oder nur schwach) über formale Rollen reglementiert sind, erfüllt hier, neben der interaktiven Beitragssélection der Beitragenden (*Peer Review*), die informelle *Mitgliedschaft* – im Sinne einer informellen Zugehörigkeit – eine Selektionsfunktion, da hierdurch der Kreis der potenziell Beitragenden definiert wird. Feedback und Arbeit an der Dokumentation kommen nahezu ausschließlich von Usern der Software selbst, die aber, wie im vorigen Kapitel gezeigt, durch die Voraussetzungen für die Mitgliedschaft sowie auch durch die Gestaltung der Software und des Wissenspools selektiert werden. Aufgrund der geringen formalen Regulie-

Die Gestaltung dieser Beitragsformen spielt hier also die informelle Selektion der Mitglieder der Community durch die Gestaltung der Software (vgl. Kap. 9) und des Wissenspools eine funktionale Rolle. Die Anforderungen an Beiträge sowie die Voraussetzungen für die formale Mitgliedschaft sind in Debian und Arch deutlich höher angesetzt als bei Ubuntu. Während bei Debian die formale Mitgliedschaft in aller Regel eine Entwicklerrolle darstellt, können bei Ubuntu explizit auch User, die keine *Contributory Expertise* im hier verwendeten Sinne besitzen, formales Mitglied der Community werden. In Ubuntu werden Beiträge von weniger erfahrenen Usern eher als valide erachtet. Im Kontrast dazu werden bei Arch auch die User implizit als aktive Entwickler*innen betrachtet, es gibt also idealiter nur User mit *Contributory Expertise*.

11.2.3 Differenzierungen von Wissen jenseits der „Interactional Expertise“

Als Nicht-Expert*innen verstehe ich User, die die Funktionsweise der Software im Sinne des Softwarecodes nicht nachvollziehen können, also weder *Contributory* noch *Interactional Expertise* besitzen. Sie besitzen dennoch in verschiedener Tiefe Wissen über die Funktionsweise der Software, von der Kenntnis der Dokumentationen und Anleitungen (*Primary Source Knowledge*) über ein generelles Verständnis der Software, wie es sich aus Zeitschriften und entsprechenden Webseiten ergibt (*Popular Understanding*), bis hin zu einem sehr rudimentären, schematischen Verständnis (*Beer-Mat Knowledge*).

Diese Gruppe von Mitgliedern spielt in den betrachteten epistemischen Regimen sehr unterschiedliche Rollen. Während in Arch nur User mit zumindest *Primary Source Knowledge* eine Chance auf eine erfolgreiche Installation der Software haben (vgl. 9.3.2), wird für eine aktive Rolle in der Community ein Mindestmaß an *Interactional Expertise* notwendig, die sich aus der Lektüre der Dokumentation (RTFM) und einer Auseinandersetzung mit dem System – angefangen bei der Installation – schrittweise ergibt.

Die Installation von Debian ist mit einem rein schematischen Verständnis (*Beer-Mat Knowledge*) von Computern zumindest schwierig, während Ubuntu explizit darauf ausgelegt ist, auch mit minimalen Kenntnissen eine Installation zu absolvieren und das System zu nutzen. Folglich werden in Ubuntu auch User mit marginalen Kenntnissen integriert und Beiträge gerin-

11.2 Die epistemischen Rekonfigurationen der Experten-Laien-Differenz 293

gerer technischer Tiefe als valide Beiträge betrachtet. Auf diese Weise werden die Perspektive der Zielgruppe „einfache User“ über das Feedback zur „User Experience“ integriert und auch einfache Fragen als wichtiger Beitrag für einfach verständliche Anleitungen und für eine einfache Bedienung des Systems betrachtet.¹⁵⁶

11.3 Spielräume zwischen Experten-Entwickler und Laien-User

Entgegen der These einer klaren Trennung von Expert*innen und Laien (vgl. Kap. 4) in der Technikentwicklung offenbaren sich in den betrachteten Communities unterschiedliche Spielräume zwischen den Expert*innen, die als Entwickler*innen die Regeln der Technik definieren, und Laien, die als User lediglich auf diese Regeln zurückgreifen, ohne näheres Verständnis der Funktionsweise des Systems.

Spielräume bedeutet hierbei nicht, dass sich die Kategorien Experte/in und Laie völlig auflösen, aber dass Expertise und Gestaltung der Technik über verschiedene Stufen ausdifferenziert sind. Einerseits sind in den betrachteten Communities die Entwickler*innen meist auch User des Systems, da die Entwicklung in der Regel intrinsisch durch die Nutzung motiviert ist, was aber noch nicht gegen die These spricht, dass sie einen Wissensvorsprung gegenüber den Laien-Usern haben (vgl. Kap. 4). Andererseits sind nicht alle User auch gleichzeitig Entwickler*innen des Systems, hier gibt es also durchaus eine Grenze zwischen Usern und Entwickler*innen; die zentrale Frage ist aber nun, wie sich die Beteiligung an der Gestaltung der Technik zur Notwendigkeit von Expertise verhält. Eine klare Grenze in Bezug auf die Veränderung der Software selbst ergibt sich zunächst durch die formalen Zugriffsbeschränkungen auf den Softwarecode; wie deutlich wurde, hat hier nicht jede*r unmittelbare Modifikationsrechte, wenngleich grundsätzlich jeder User Änderungen vorschlagen und Code-Beiträge anbieten darf. Diese

156 Zumindest ist dies das erklärte Programm und wird auch in den Interviews so dargestellt, was freilich nicht heißt, dass es innerhalb der Community nicht auch Zurückweisungen von Fragen und RTFM gibt – ebenso, wie es in Arch mitunter geduldige Hilfestellung für einfache Fragen gibt.

Grenzziehung zwischen Beitragenden und Nicht-Beitragenden geschieht über die Selektion von Beiträgen, vermittelt durch die epistemischen Regime der Communities.

Durch die Rekonfiguration der Experten-Laien-Differenz ergeben sich zweierlei Unterscheidungen: die Unterscheidung von Entwickler*innen und Nicht-Entwickler*innen im Sinne einer *Contributory Expertise* (als Beitrag zum Softwarecode) einerseits und andererseits die Unterscheidung zwischen Expert*innen, die nicht direkt zum Softwarecode beitragen, aber *Interactional Expertise* über die Funktionsweise der Software besitzen, und den Nicht-Expert*innen, die sich auch mit der Funktionsweise auseinandersetzen, aber nicht die *Interactional Expertise* erlangen, um sich mit Entwickler*innen auf Augenhöhe auszutauschen.

Die Grenze ist hierbei nicht trennscharf zu ziehen, da die Auseinandersetzung mit der Dokumentation der Software und die Auseinandersetzung mit der Software selbst, zumindest auf der Ebene der Kommandozeile, eine grundlegende Erfahrung der fundamentalen Programm-Logiken wie Input-Output-Relationen, Parameter-Übergabe und Exit-Status impliziert (vgl. S. 31). Auch enthalten die Dokumentations-Einträge auf der Ebene der Kommandozeile (sogenannte *Man-Pages*) Referenzen zu verwandten Befehlen und ein tieferer Einstieg in das Zusammenspiel des Betriebssystems ist darin angelegt.

Es gibt also eine formale Trennung in Bezug auf den direkten Schreibzugriff auf die Software, aber es gibt Spielräume in der Rolle von Expertise für verschiedene Beitragsformen im vermittelten Einfluss auf die Gestaltung der Software und in der organisatorischen Gestaltung der formalen Rollen.

11.3.1 Entwickler-User-Differenz

Im Folgenden werden die verschiedenen Ausgestaltungen der Entwickler-User-Differenz von Fall zu Fall skizziert. Das Augenmerk liegt dabei auf der Gestaltung der Rollen der formalen Organisation der Community. Dabei zeigt sich bei Ubuntu eine stärkere Rollendifferenzierung, mit niederschweligen Rollen, die auch explizit Laien-User einschließt, die nicht am Softwarecode entwickeln können. Letztere haben keinen direkten Einfluss auf den Softwarecode, wohl aber einen indirekten über die Mitarbeit in der Community. Demgegenüber steht eine schwache Rollendifferenzierung bei Debian, die eine hohe Eintrittsschwelle aufweist und sich auf die Entwicklungstätig-

keiten konzentriert. Arch hingegen hat keine formalen User-Rollen, allerdings können User Entwickleraufgaben übernehmen. Die Entwicklerrollen haben aber eine starke Differenzierung und die Rolle der Core-Developer ist nicht durchlässig, sondern die Auswahl neuer Mitglieder erfolgt allein durch interne informelle Absprache.

Ubuntu: Ausdifferenzierte Entwickler- und User-Rollen

In Ubuntu wird zwischen *Ubuntu Developers* und *Ubuntu Members* unterschieden, von denen letztere auch eine formale Rolle innehaben. Die Entwicklerrolle ist relativ niederschwellig, insofern es auch reduzierte Entwicklerrollen gibt, deren Rechte auf enge Bereiche eingegrenzt sind, und jeder User auch ein Software-Paket im persönlichen Repository („Personal Package Archive“) bereitstellen kann, über das andere User auf die Software zugreifen können. Der Zugriff auf die Kernbereiche der Software ist aber höher privilegierten Entwickler*innen vorbehalten (Core-Developer).

Durch die formale Rolle des Users haben diese zwar keinen direkten Einfluss auf die Gestaltung der Software, können aber in der Community sichtbar werden und mittelbaren Einfluss nehmen. Die Rolle hat aber offenbar einen stark symbolischen Charakter, da aus der Rollenbeschreibung nicht klar ersichtlich ist, welche Entscheidungsrechte mit der Rolle einhergehen – jenseits einer offiziellen Anerkennung und einer Stimme für die Bestätigung des *Community Council* (vgl. Abschnitt 8.1.1).

*Debian: Alle Entwickler*innen sind gleich*

Bei Debian gibt es eine klare, formale Trennung zwischen Entwickler*innen und Usern. Letztere haben keine formale Rolle inne, können also nur vermittelt über die Entwickler*innen oder durch Erlangung einer Entwicklerrolle zum Code beitragen. Anerkannte Entwickler*innen haben Entscheidungsrechte bei der Wahl des Projektleiters oder der Projektleiterin und bei demokratischen Abstimmungen über die sogenannte *General Resolution*. Die Erreichung der Entwicklerrolle ist aber über den Prozess des vermittelten Beitragens angelegt und ist somit prinzipiell jedem zugänglich, der entsprechende Kompetenzen und entsprechendes Engagement vorweisen kann.

Die Möglichkeit, auch ohne Entwicklertätigkeit Teil der formalen Organisation zu werden, gibt es seit geraumer Zeit zwar auch, spielt aber noch eine marginale Rolle (vgl. S. 165). Es gibt also offensichtlich Bestrebungen dahingehend, Nicht-Entwickler*innen in die Organisation einzubinden, aber eine

wirksame Veränderung der sozialen Struktur ist schwer zu erreichen. Das Regime zeigt sich hier recht stabil.

Arch: Entwickler-Elite und User-Entwickler

Bei Arch wird einerseits jeder User auch als Entwickler*in betrachtet und die User-Rolle ist mit Entwickleraufgaben bedacht. Dennoch gibt es eine starke Unterscheidung zwischen dem Kernbereich und einem Community-Bereich. Die Unterscheidung ist eine formale und auch eine nominale Differenzierung. Die Kernbereiche verwalten die *Arch Developer*, die Entwickleraufgaben im Bereich der Community übernehmen sogenannte *Trusted User*. Es handelt sich also im Grunde um eine Gemeinschaft von Entwickler*innen, in denen aber eine kleine Elite die Hoheit über die wichtigen Bereiche und die Entscheidungen hat.

*Entwickler*innen: Die zentrale gestaltende Rolle*

Es zeigt sich, dass es sehr unterschiedliche Differenzierungen der Entwicklerrollen gibt und sich hier je Community unterschiedliche Einflussmöglichkeiten innerhalb einer formalen Rolle in der Organisation ergeben.

Besonders interessant ist die formale Integration der User in Ubuntu und ansatzweise auch in Debian. Allerdings hat diese bei Ubuntu einen stark symbolischen Charakter. Die formale Anerkennung spielt aber im Zusammenspiel der verschiedenen Variablen des epistemischen Regimes dennoch eine wichtige Rolle. Bei Debian bleibt die formale User-Rolle zunächst außen vor. Unter dem Label von Entwickler*innen ohne Schreibrechte wird dies aber im Ansatz abgebildet. Diese Rolle bleibt zwar marginal, aber abgesehen von den Schreibrechten für die Software ist sie bezüglich der Mitbestimmung im Rahmen der Organisation (Projektleiterwahl und demokratische Abstimmungen) gleichberechtigt. Die geringe Verbreitung der Rolle könnte aber damit zusammenhängen, dass die Schwelle für ein erfolgreiches Bewerbungsverfahren recht hoch ist und dass sich die Praxis der Inklusion von Nicht-Entwickler*innen noch nicht etabliert hat.¹⁵⁷

157 Im Laufe der Finalisierung der Arbeit stelle ich fest, dass die Möglichkeit eines Entwicklerstatus ohne Entwicklertätigkeit mittlerweile prominent auf der entsprechenden <https://www.debian.org/devel/join/newmaint> (letzter Aufruf: 16.6.2018) genannt wird, was zu Beginn der Arbeit noch nicht so war. Offenbar gibt es Versuche, diese Form der Mitgliedschaft stärker zu etablieren.

Eine zentrale Rolle der Entwickler*innen ist vor dem Hintergrund der technischen Ausrichtung der Entwicklung von Software nicht weiter verwunderlich. Bemerkenswert sind umso mehr die Effekte der formalen Organisation und der Differenzierung verschiedener Wissenslevel innerhalb der Community auf das Zusammenspiel der Variablen als epistemische Regime.

11.3.2 Experten-Laien-Differenz

Im Gegensatz zur Grenzziehung zwischen Entwickler*innen, die direkten Zugriff auf die Regeln der Technik haben, und Nicht-Entwickler*innen, die lediglich vermittelt Einfluss darauf nehmen können, ist das Verhältnis zwischen Expert*innen und Laien stärker ausdifferenziert. In meiner Konzeption des Begriffs habe ich das Verständnis von Expertise erweitert. Expert*innen sind dabei nicht ausschließlich die direkten *Gestalter*innen* der Regel, sondern neben den Entwickler*innen mit *Contributory Expertise* gibt es auch andere Expert*innen, die *Interactional Expertise* besitzen. *Laien* sind einige Wissensstufen darunter zu verorten, der Begriff Laie scheint dabei nicht ganz passend, da er eine Dichotomie nahelegt. Vielmehr geht es hier um unterschiedlich tiefes Wissen, das verschiedene Akteure besitzen. Die Bedeutung dieser Kategorien von Wissen – beziehungsweise die Rolle von Beitragenden, die unterschiedliches Wissen besitzen und nicht *Interactional* oder *Contributory Expertise* haben – macht den Unterschied in den betrachteten Fällen aus und lässt sich als unterschiedliche Experten-Laien-Differenzen beschreiben.

Zwar ist in jedem Fall ein höherer Grad an Wissen hilfreich, um Beiträge beizusteuern, die integriert werden und somit das gemeinsam produzierte Wissen gestalten, jedoch unterscheiden sich die Fälle stark darin, inwieweit Beiträge von Akteuren mit geringeren Graden an Wissen akzeptiert und bearbeitet werden. Dies gilt weniger für das Angebot konkreter Softwarecode-Änderungen, sondern insbesondere für den Bereich der interaktiv vermittelten Beiträge in Form von Feedback, Dokumentation und Support. Hier ist das Verständnis von Softwarecode nicht notwendig, wenn auch hilfreich. Ubuntu fällt hier aber besonders auf, weil eben die Perspektive von weniger erfahrenen Usern als hilfreich für die Ausrichtung der Wissensprodukte auf eben diese Nutzergruppe erachtet wird. Folglich spielen hier Beiträge von Laien eine besondere Rolle.

Da die Beiträge üblicherweise von Usern erbracht werden, wirkt die produzierte Software sowie der Wissenspool nicht nur als Spiegel der Werte der Community, sondern als Selektor für neue Mitglieder der Community und hat damit indirekt einen Einfluss auf potenzielle Beiträge. Im Folgenden gehe ich daher zunächst von der Wirkung der Software auf die User aus (vgl. Kap. 9), um anschließend die Rolle verschiedener Wissenslevel zu skizzieren.

Ubuntu: Laien-Inklusionsregime

Bei Ubuntu zeigt sich beim oben beschriebenen Installationsprozess, dass die Software explizit darauf ausgelegt ist, auch für User mit wenig Wissen über die Funktionsweise von Computern installierbar und bedienbar zu sein. Dementsprechend sind auch Beiträge von Laien – in Form von einfachen Fragen und Feedback über die Funktionalität der Software in puncto Verständlichkeit – willkommen. Die Zugänglichkeit der Software führt einerseits zu mehr Beiträgen dieser Nutzergruppe und andererseits strukturieren deren Beiträge die Software.

Für das Beitragen reicht an vielen Stellen schon ein schematisches Verständnis des Systems aus (*Beer-Mat Knowledge*). Im Gegensatz zu Arch sind auch einfache Fragen im Supportbereich grundsätzlich erlaubt. Aber auch im Bereich der Feedback-Schleife für die Entwickler*innen sind grundsätzlich auch einfache Verständnisfragen der Benutzerführung erlaubt. Darüber hinaus werden unvollständige Fehlerberichte tendenziell mit mehr Wohlwollen aufgenommen und es gibt sogar einen integrierten automatischen Fehlerbericht. In der Community spielen dann „Interactional Experts“ eine wichtige Rolle für die Übersetzung der Beiträge der User für die Entwickler*innen. Gerade bei Fehlerberichten ist es notwendig, doppelte Fehlerberichte auszusortieren und fehlende Informationen für die Bearbeitung des Fehlers zu erfragen oder durch eigenes Testen herauszufinden. Dies wird auch Bug Triaging genannt, eine Beitragsform, die in den anderen Interviews keine Erwähnung fand (vgl. Abschnitt 8.3.1).

Dadurch gibt es für Laien eine explizite Rolle in der Gemeinschaft, als reine*r Nutzer*in einerseits und als beitragendes Mitglied andererseits. Schließlich kann sich ein User auch durch Beiträge, die nicht auf Ebene der Code-Entwicklung sind, als offizielles Ubuntu-Mitglied qualifizieren. Somit birgt die formale Struktur der Community ein Inklusionspotenzial für Laien.

Mit anderen Worten ist in den Bereichen, in denen es nicht um den Beitrag von Softwarecode geht, die Akzeptanz von Beiträgen auch geringerer Wissenslevel deutlich höher als bei den anderen beiden Fällen. Dies ist der

expliziten Orientierung auf deren Bedürfnisse geschuldet und wird durch die organisationalen Strukturen gestützt. Eine Integration der Perspektive von einfachen Usern und deren Beitrag zur Gestaltung des Wissensprodukts ist also explizit gewollt. Dennoch bleibt der Beitrag auf der vermittelten Ebene, denn die eigentliche Entwicklung der Software geschieht auch hier durch die *Contributory Experts*. Durch die Konfiguration des epistemischen Regimes wird aber eine Gestaltung der Software im Sinne auch weniger technisch versierter User strukturell unterstützt.

Debian: Traditionell expertenlastig

Im Installationsprozess von Debian zeigt sich zwar eine gewisse Adressierung von Laien in den Empfehlungen von Vorkonfigurationen für Anfänger*innen, dennoch gibt es auch hier für Laien einige Hürden und der Installationsprozess birgt eine gewisse Notwendigkeit der Aneignung von Wissen über das System.

Ein gewisser Wille zur Inklusion von Einsteiger*innen ist also durchaus gegeben, aber die Umsetzung gelingt nicht im selben Maße wie bei Ubuntu, hat aber auch nicht dieselbe Priorität. In Debian gestaltet sich die Beteiligung von einfacheren Usern eher schwierig. Es gibt strenge Policies, die bei Fehlerberichten beachtet werden sollen und deren Einhaltung stark eingefordert wird, und die Tools dafür sind laut Interview-Aussagen weniger leicht zu bedienen als beispielsweise bei Ubuntu. Mit *Beer-Mat Knowledge* kommt man hier also nicht weit, eher sollte man *Interactional Expertise* mitbringen, um sinnvoll beitragen zu können.

Offensichtlich erschwert die geringere Beteiligung von weniger erfahrenen Usern auch die Integration ihrer Perspektive. Möglicherweise gibt es aber auch weniger Entwickler*innen, die Interesse daran haben, diese Perspektive zu integrieren, als in Ubuntu, wo es einen Stab von bezahlten Canonical-Entwicklern gibt, die auch Dinge entwickeln, die nicht ihrer Leidenschaft entsprechen, nutzerfreundliche Dinge zum Beispiel. Beides ist Teil der Konfiguration des epistemischen Regimes und führt dazu, dass – trotz einem erkennbaren Willen zu mehr Usability – das Regime im Fall von Debian expertenlastig bleibt.

Der Wille zur formalen Integration von Usern, die keine *Contributory Experts*, Entwickler*innen, sind, zeigt sich auch in der Ausweitung der formalen Mitgliedschaft als *Debian Developer* auf *non-uploading* Developers. Die ausbleibende Etablierung dieser Rolle zeigt aber (vgl. letzter Abschnitt und S. 165), dass die Einrichtung dieser Rolle allein noch keine Veränderung des

epistemischen (Experten-)Regimes bewirkt. Die *non-uploading* Developer müssen aber offenbar ebenso wie die anderen *Debian Developer* weitreichende Kenntnisse über Debian und die internen Projektprozesse vorweisen. In dieser Hinsicht ist die Hürde für die Mitgliedschaft weiterhin hoch angesiedelt und ein Brückenschlag zu den Nicht-Expert*innen bleibt aus.

Arch: Explizite Expertengemeinschaft

Die Arch-Installation setzt als Messlatte für User ein Mindestmaß an Systemwissen an, für eine erfolgreiche Installation ist eine Auseinandersetzung mit der Dokumentation zwingend erforderlich. Damit ist Arch darauf angelegt, dass User zu Expert*innen ihres Systems werden und bei Bedarf auch in die Entwickler-Rolle schlüpfen. Arch ist also als Expertengemeinschaft angelegt. *Laie* wird hier allenfalls als Übergangsrolle verstanden. Einfache Fragen widersprechen dem Anspruch, sich selbst mit der Materie auseinanderzusetzen und durch das Studium der Dokumentation und viel Ausprobieren selbst die Lösungen zu erarbeiten. Daher sind unterhalb der *Primary Source Knowledge* keine Beiträge und keine User vorgesehen.

Durch die Auseinandersetzung mit den eigenen Problemen erlangen auch Einsteiger*innen nach und nach *Interactional Expertise*, die Community kann also in dieser Hinsicht als Experten-Community betrachtet werden. Folglich ist es auch nicht notwendig, weniger qualifizierte Beiträge zu akzeptieren oder die Perspektive von Laien zu integrieren.

In dieser Logik ist es auch unhinterfragt akzeptiert, dass die Steuerung des Projekts durch eine kleine Elite von Arch-Veteranen geschieht. Einerseits haben diese sich das durch ihre Leistung und ihr Können „verdient“ und andererseits gibt es aus einem technischen Verständnis und dem klaren Konsens von „Keep It Simple, Stupid“ heraus offenbar auch wenig zu diskutieren. Die Exklusion von Laien ist aber hier eine bewusste Entscheidung zur Steigerung der Effizienz des Projekts und zur Fokussierung auf das Wesentliche, nämlich die technische Weiterentwicklung.

Laien- und Expertenregime

Während auf der reinen Entwicklungsebene in allen Fällen Entwickler*innen, *Contributory Experts*, zentral sind, mischt sich das Feld in den weniger formalisierten Bereichen des Beitrags. Bemerkenswert ist, dass, obwohl rein formal und auch entwicklungspraktisch Laien-Nutzer*innen wenig Einfluss haben, die Perspektive von Usern – jene, die eher keine Expert*innen sind – doch sehr deutlich durch die Gestaltung der Community,

durch die Konfiguration des epistemischen Regimes, beeinflusst wird und praktische Auswirkungen auf die Wissensprodukte zeigt.

Es zeigt sich, dass diese Wirkung auf allen Variablen der epistemischen Regime fußt und erst durch das Zusammenspiel als epistemische Regime das Wissen auf die eine oder andere Art geformt wird. Das Wechselspiel der formalen Organisation, der formalen Mitgliedschaft sowie der informellen Identifikation mit den Werten der Community (*Selbstbild der Community*) und der praktischen interaktiven Gestaltung der Beiträge formt das gemeinsam produzierte Wissen.

Die verschiedenen Variablen der epistemischen Regime wirken über ihre informelle identitätsstiftende Funktion und die formale Legitimation der verschiedenen entscheidenden Akteure und das Vertrauen der User in die Software, worauf ich im abschließenden Kapitel näher eingehen werde.

12 Schlussfolgerungen

Epistemische Regime regulieren Legitimation und Vertrauen

Welche Folgerungen ergeben sich für die zu Anfang aufgeworfenen Probleme der Legitimation, der Frage nach Vertrauen in die Funktionsprinzipien der Software und in Bezug auf die Macht der Entwickler*innen, die sich in der Sachtechnik sonst üblicherweise als Dichotomie zwischen Expert*innen und Laien darstellt?

Wie im letzten Abschnitt deutlich wurde, ergeben sich verschiedene Relationen zwischen Expert*innen und Laien und zwischen Entwickler*innen und Usern. Dabei gibt es formal weiterhin eine wirksame Unterscheidung zwischen der Pflege der technischen Regeln, dem Code der Software, durch Entwickler*innen und dem angewandten Zugriff auf diese Regeln durch User. Allerdings variiert die soziale Ordnung der Entwickler*innen und darüber hinaus differenziert sich das notwendige Wissen für verschiedene Formen der Beteiligung an der Produktion gemeinsamen Wissens. Jenseits der Entwicklung am funktionalen Software-Teil fallen die Anforderungen an den Beitrag zur Wissensproduktion auseinander. Dies wird formal und informell vermittelt durch die epistemischen Regime der Communities und strukturiert das gemeinsame Wissen. Das Wissensprodukt selbst wirkt schließlich wieder strukturierend zurück auf die Community.

*Legitimation von und Vertrauen in die Autorität der Expert*innen*

Es ist deutlich geworden, dass auch in den betrachteten FLOSS-Communities nicht jeder User Zugriff auf die Veränderung des Softwarecodes hat, der für alle die Basis ihres Betriebssystems bildet. Vielmehr bilden die epistemischen Regime ein Geflecht von Entscheidungsmodalitäten, Zuständigkeiten, Rechten und Aufgaben und schließlich Kriterien für die Bewertung von Beiträgen und für die Zuweisung von Rollen.

Folglich gibt es sichtbare Strukturen und Regeln, die die Verfassung des Regimes beschreiben und somit nachvollziehbar machen, welche Kriterien und Verfahren den Entscheidungen zugrunde liegen und somit ein gewisses Vertrauen stiften. Die Machtrelationen sind hierbei jedoch unterschiedlich gelagert, teils transparenter oder intransparenter, teils auch eher implizit und informell. Ein großer Teil der Entscheidungen geschieht dezentral und ad hoc direkt in der Entwicklung und kann lediglich durch die Betrachtung der Do-

kumentation der Veränderungen („Logs“) nachvollzogen werden, andere Entscheidungen werden stärker diskutiert und manche werden unilateral, konsensual oder demokratisch abgestimmt.

Das Zusammenspiel der formalen Organisation durch Strukturen und Prozesse der Community mit Mitgliedschaft als kollektivem Selbstbild und schließlich die praktische Gestaltung der Beiträge zum kollektiven Wissen formen den Möglichkeitsraum des produzierten Wissens und schreiben die Normen der Community darin ein. Die Rahmung als partizipative Gemeinschaft, die eigenen definierten Regeln folgt, motiviert dabei ein Vertrauen in die Richtigkeit gemeinsamer Prinzipien.

Während die Strukturierung des Wissens schwierig auf einen Punkt zu verorten ist und die Laien weder formell noch informell *starke* Einflussmöglichkeiten haben, zeigt sich doch in der formalen Organisation eine wirksame Legitimierung der Autorität der Entwickler*innen und der entscheidenden Organe und Mitglieder. Die Legitimation basiert dabei auf je spezifischen Kriterien:

- Debian basiert auf einem stark bürokratischen Aufnahmeverfahren, das definierte Wissensbestände bei Bewerber*innen abfragt, und gibt den erfolgreichen Mitgliedern gleiche Rechte.
- Ubuntu legt einen Schwerpunkt auf die symbolische Integration von Laien durch die User-Mitgliedschaft und spezifische Organe, die dem Community-Management gewidmet sind.
- Arch schließlich legitimiert ihre Führungselite durch technische Erfahrung und „Überlegenheit“.

Die Legitimierung der Autorität der Rollen definiert sich also durch formale Strukturen und basiert je Community auf spezifischen inhärenten Werten, die die Kriterien der Legitimität begründen.

Die organisational legitimierte Autorität der gestaltenden Akteure stiftet wiederum Vertrauen in die Richtigkeit der Zuweisung von Rollen an bestimmte Individuen sowie die Richtigkeit ihrer Entscheidungen bezüglich der Entwicklung der Software und der Weiterentwicklung des gemeinsamen Wissens.

Legitimität und Vertrauen sind dabei freilich nicht für allezeit stabil, sondern sind auch Gegenstand von internen Diskussionen und Aushandlungsprozessen. Zentral sind dabei aber das Zugehörigkeitsgefühl und die Gruppenidentität, die durch die formalen Mitgliedschaften geprägt sind, aber auch auf der Selbstzuordnung und dem Selbstverständnis der User basieren. Auch

die User, die nicht formale Mitglieder sind, tragen einen wesentlichen Teil zu den Diskursen über die Software und die Gestaltung des Wissens bei – durch kommunikative Interaktionen etwa, aber auch durch Beiträge in den informell regulierten Bereichen.

Freie Software und die Ausdifferenzierung epistemischer Regime

Die Werte der Communities sind manifest in der Präsentation des Projekts, in den grundlegenden Dokumenten (Policies und *Code of Conduct*) und schließlich in der Gestaltung der Software. User haben damit die Möglichkeit, zu erfahren, auf was sie sich einlassen, können entscheiden, wie tief sie sich darauf einlassen, und haben grundsätzlich die Möglichkeit, sich auch zu beteiligen. Dadurch haben User auch eine Wahlmöglichkeit zwischen verschiedenen Communities und darauf, welche Werte sie präferieren und welche Kriterien dem Entwicklungsprozess zugrunde liegen und sich schließlich in der Software widerspiegeln.

Erst die Öffnung des Quellcodes ermöglicht die Ausbildung verschiedener epistemischer Regime, die funktional äquivalente technische Systeme produzieren, die aber Unterschiede aufweisen in Nutzung, Konfiguration und Programmierung und eine Varianz an Wertekonfigurationen der Produktentwicklung und damit der in das Produkt eingeschriebenen Werte.

Erst recht spät entwickelte sich dabei ein Regime heraus, das explizit Laien einen Zugang ermöglicht und auch den Verbleib in einer User-Rolle vorsieht. Dafür wurden aber intensiv Ressourcen investiert und der Gründer von Ubuntu hat einige Innovationen gegen Widerstände in der FLOSS-Community durchgesetzt (vgl. die Diskussionen um die grafische Oberfläche *Unity*, S. 159). Mit dem Anspruch eines „Linux for Human Beings“ hat Ubuntu aber nicht zuletzt auch nachhaltige Effekte auf andere Linux-Communities ausgeübt, wie man an den Ansätzen der Integration von Einsteiger*innen (bspw. im Installer, Abschnitt 9.2) und Nicht-Entwickler*innen (S. 165) bei Debian sehen kann.

Schlussbemerkungen und offene Fragen

Dennoch bleibt der Einfluss der Einzelnen beschränkt. Wer nicht programmieren kann, bleibt angewiesen auf offene Ohren für seine oder ihre Vorschläge. Je besser man die Sprache der Entwickler*innen sprechen kann (*Interactional Expertise*), desto besser kann man sie überzeugen von der Notwendigkeit einer Funktion und der Dringlichkeit eines Fehlers.

Zudem gestaltet sich die Integration der User-Perspektive bisweilen schwierig. Wir haben im Fall von Ubuntu gesehen, dass hier relativ viel Aufwand getrieben wird. Aber hier spielt Lohnarbeit eine wichtige Rolle, denn für Community-Management und die Entwicklung der Oberflächen, der User Experience, werden Angestellte bezahlt. Bereiche also, die üblicherweise weniger intrinsische Anliegen der FLOSS-Communities sind, werden hier über FLOSS-untypische Strukturen bearbeitet.

Es bleibt aber die strukturelle Transparenz der Software und auch der sozialen Struktur ihrer Herstellung. Diese trägt zwar klare Wertungen in sich, ist aber vergleichsweise sichtbar und innerhalb der Community verhandelbar. Im Zweifelsfall lässt sich ein neues Projekt abspalten und mit veränderten Prämissen weiterführen – wenngleich die Einschreibungen des abgespaltenen Codes zunächst weiter existieren. In Anbetracht der wachsenden Bedeutung von Software birgt dies aber ein demokratisches Potenzial.

Es hat sich gezeigt, welche zentrale Rolle die Organisationsstrukturen der Communities spielen. Diese werden von den Mitgliedern beeinflusst und beeinflussen die Zusammensetzung der Community und die Bewertung der Beiträge, was wiederum das produzierte Wissen formt. Die betrachteten Fälle sind aber nur ein kleiner Ausschnitt der Softwarewelt.

Für ein tieferes Verständnis der normativen Strukturierung der Softwareproduktion und ihres Einflusses auf ihre Anwendung ist weitere Forschung notwendig. Wichtig wäre hierfür, sich aus der Welt des Betriebssystems herauszubewegen und einen Blick in andere Sachtechniken eingebetteter Systeme zu werfen: Autos, Industriemaschinen und tragbare Geräte. Lassen sich hier in ähnlicher Weise normative Konfigurationen der sozialen Strukturen der Herstellung finden? Lassen sich diese auch als epistemische Regime fassen und eingeschriebene Wertesysteme herausarbeiten? Welche epistemischen Strukturen finden sich in Unternehmen?

Eine besondere Bedeutung kommt der Betrachtung von algorithmischen Entscheidungssystemen zu beziehungsweise Systemen, in denen nicht der Algorithmus, sondern die Trainingsdaten Ausgangspunkt normativer Einschreibungen sind. Sind hierbei ähnliche Strukturierungen nachweisbar? Wie wird hier Legitimität konstruiert? Dies sind Fragen, deren Beantwortung dazu beitragen kann, eine politische Gestaltung von Software zu diskutieren, denn die eingeschriebenen und wirksamen Werte sollten Gegenstand gesellschaftlicher Aushandlung sein und nicht implizit und unhinterfragt bleiben.

Danksagung

Die vorliegende Arbeit wurde ermöglicht durch die Unterstützung der Konrad-Adenauer-Stiftung im Rahmen eines Promotionsstipendiums.

Besonderer Dank gilt den Kolleginnen und Kollegen, die mich in meiner Arbeit begleitet haben: Ingo Schulz-Schaeffer für die strukturierte Betreuung und Unterstützung, Jochen Gläser für die konzeptionelle Beratung von Anfang an und die stetige Begleitung des Projekts. Dank gilt auch Ingo Schulz-Schaeffers Doktorandenkolloquium an der Universität Duisburg, den Kolleginnen und Kollegen an der Technischen Universität Berlin und dem Zentrum Technik und Gesellschaft für gedanklichen Austausch und dem „Berlin Script Collective“ für das spannende und fruchtbare Projekt der Entwicklung eines Technikvergleich-Rahmens, der über die Arbeit hinaus Anknüpfungspunkte bietet.

Großer Dank gilt meiner Familie für alle Unterstützung, Nachsicht und Geduld.

Schließlich möchte ich der Free/Libre Open Source Software Community danken für das politische Engagement und für wunderbare Software, die meinen Rechner nutzbar macht und mir die tägliche Arbeit ermöglicht, und der gesamten netzpolitischen Bewegung, die sich dafür einsetzt, dass Informationstechnik Freiräume gewährt anstatt sie zu beschneiden.

Literaturverzeichnis

- Akrich, Madeleine (1992): The De-Description of Technical Objects. In: *Shaping Technology/Building Society*. Hrsg. von Wiebe E. Bijker; John Law. Cambridge, London: MIT Press, S. 205–224.
- Anderson, Christopher W. (2012): The Materiality of Algorithms. In: *Culture Digitally*. URL: <http://culturedigitally.org/2012/11/the-materiality-of-algorithms/> (besucht am 03. 10. 2018).
- Bagozzi, Richard P.; Utpal M. Dholakia (2006): Open Source Software User Communities: A Study of Participation in Linux User Groups. In: *Management Science* 52.7, S. 1099–1115. <https://doi.org/10.1287/mnsc.1060.0545>.
- Barcellini, Flore; Françoise Détienne; Jean-Marie Burkhardt (2008): User and Developer Mediation in an Open Source Software Community: Boundary Spanning through Cross Participation in Online Discussions. In: *International Journal of Human-Computer Studies* 66.7, S. 558–570.
- (2009): Participation in Online Interaction Spaces: Design-Use Mediation in an Open Source Software Community. In: *International Journal of Industrial Ergonomics* 39.3, S. 533–540.
- Barcellini, Flore; Françoise Détienne; Jean-Marie Burkhardt; Warren Sack (2005): A Study of Online Discussions in an Open-Source Software Community. In: *Communities and Technologies 2005*. Hrsg. von Peter van den Besselaar u. a. Dordrecht: Springer, S. 301–320.
- Barocas, Solon; Sophie Hood; Malte Ziewitz (2013): Provocation Piece. Governing Algorithms Conference Website. URL: <https://governingalgorithms.org/resources/provocation-piece/> (besucht am 03. 10. 2018).
- Berlin Script Collective (2018): Technik vergleichen: Ein Analyserahmen für die Beeinflussung von Arbeit durch Technik. In: *Arbeits-und Industriesoziologische Studien* 11.2, S. 124–142.
- Berry, David M. (2008): *Copy, Rip, Burn. The Politics of Copyleft and Open Source*. London: Pluto Press.
- Blumenberg, Hans (1981): Lebenswelt und Technisierung unter Aspekten der Phänomenologie. In: *Wirklichkeiten, in denen wir leben. Aufsätze und eine Rede*. Stuttgart: Reclam, S. 7–54.
- Böschen, Stefan (2016): *Hybride Wissensregime. Skizze einer soziologischen Feldtheorie*. Baden-Baden: Nomos.

- Brownsword, Roger (2005): Code, Control, and Choice: Why East is East and West Is West. In: *Legal Studies* 25.1, S. 1–21.
- Chun, Wendy Hui Kyong (2011): *Programmed Visions: Software and Memory*. Cambridge, London: MIT Press.
- Coleman, E. Gabriella (2013): *Coding Freedom: The Ethics and Aesthetics of Hacking*. Princeton [u.a.]: Princeton University Press.
- Collins, Harry M.; Robert Evans (2002): The Third Wave of Science Studies: Studies of Expertise and Experience. In: *Social Studies of Science* 32.2, S. 235–296. <https://doi.org/10.1177/0306312702032002003>.
- (2007): *Rethinking Expertise*. Chicago [u.a.]: University of Chicago Press.
- Creemers, Niklas; Daniel Guagnin; Bert-Jaap Koops (Hrsg.) (2015): *Profiling Technologies in Practice: Applications and Impact on Fundamental Rights and Values*. Oisterwijk: Wolf Legal Publishers.
- Davidson, Jennifer L.; Umme Ayda Mannan; Rithika Naik; Ishneet Dua; Carlos Jensen (2014): Older Adults and Free/Open Source Software: A Diary Study of First-Time Contributors. In: *Proceedings of the International Symposium on Open Collaboration, OpenSym '14*. ACM Press, S. 1–10.
- Davis, Martin (1985): *Computability and Unsolvability*. New York: Dover Publications.
- Degele, Nina (2000): *Informiertes Wissen: eine Wissenssoziologie der computerisierten Gesellschaft*. Frankfurt/M. [u.a.]: Campus.
- Degele, Nina (2002): *Einführung in die Techniksoziologie*. München: Wilhelm Fink Verlag.
- Divitini, Monica; Letizia Jaccheri; Eric Monteiro; Hallvard Trætteberg (2003): Open Source Processes: No Place for Politics. In: *Proceedings of ICSE 2003 Workshop on Open Source*, S. 39–44.
- Dobusch, Leonhard (2008): *Windows versus Linux: Markt – Organisation – Pfad*. Wiesbaden: VS Verlag für Sozialwissenschaften [Zugl.: Diss., Freie Univ. Berlin, 2008]
- Ducheneaut, Nicolas (2005): Socialization in an Open Source Software Community: A Socio-Technical Analysis. In: *Computer Supported Cooperative Work (CSCW)* 14.4, S. 323–368. <https://doi.org/10.1007/s10606-005-9000-1>.
- Economides, Nicholas; Evangelos Katsamakos (2006): Two-Sided Competition of Proprietary vs. Open Source Technology Platforms and the Implications for the Software Industry. In: *Management Science* 52.7, S. 1057–1071. <https://doi.org/10.1287/mnsc.1060.0549>.

- Edwards, Kasper (2001): Epistemic Communities, Situated Learning and Open Source Software Development. In: *Cultures and the Practice of Interdisciplinarity. Workshop at NTNU*, S. 11–12.
- English, William K.; Douglas C. Engelbart; Melvyn L. Berman (1967): Display-Selection Techniques for Text Manipulation. In: *IEEE Transactions on Human Factors in Electronics* 1, S. 5–15.
- Fleck, Ludwik (1979): *Genesis and Development of a Scientific Fact*. Chicago: University of Chicago Press.
- Free Software Foundation (2013): The Free Software Definition. <http://www.gnu.org/philosophy/free-sw.html> (besucht am 08.06.2009).
- Ghosh, Rishab Aiyer u. a. (2006): Study on the Economic Impact of Open Source Software on Innovation and the Competitiveness of the Information and Communication Technologies (ICT) Sector in the EU. Techn. Ber. Contract ENTR/04/112. European Commission.
- Giddens, Anthony (1984): *The Constitution of Society: Outline of the Theory of Structuration*. Berkeley [u. a.]: University of California Press.
- Giddens, Anthony (1996): Leben in einer Posttraditionalen Gesellschaft. In: *Reflexive Modernisierung*. Hrsg. von Ulrich Beck; Anthony Giddens; Scott Lash. Frankfurt/M.: Suhrkamp, S. 113–194.
- Gillespie, Tarleton (2016): Algorithm. In: *Digital Keywords*. Hrsg. von Benjamin Peters. Princeton, Oxford: Princeton University Press, S. 18–30.
- Gläser, Jochen (2006): *Wissenschaftliche Produktionsgemeinschaften: Die soziale Ordnung der Forschung*. Frankfurt/M., New York: Campus.
- Gläser, Jochen; Grit Laudel (2008): *Experteninterviews und qualitative Inhaltsanalyse als Instrumente rekonstruierender Untersuchungen*. 3., überarb. Aufl., Wiesbaden: VS Verlag für Sozialwissenschaften.
- Graham, Stephen; David Wood (2003): Digitizing Surveillance: Categorization, Space, Inequality. In: *Critical Social Policy* 23.2, S. 227–248.
- Grasmuck, Volker (2004). *Freie Software: Zwischen Privat- und Gemeineigentum*. 2. korr. Aufl., Bonn: Bundeszentrale für politische Bildung.
- Haas, Peter M. (1992): Introduction: Epistemic Communities and International Policy Coordination. In: *International Organization* 46.1, S. 1–35. URL: https://www.cambridge.org/core/product/identifier/S0020818300001442/type/journal_article (besucht am 14.08.2019).
- Heintz, Bettina (1993): *Die Herrschaft der Regel: zur Grundlagengeschichte des Computers*. Frankfurt/M., New York: Campus [Zugl.: Diss., Univ. Zürich, 1991/92].

- Hildebrandt, Mireille (2008a): Ambient Intelligence, Criminal Liability and Democracy. In: *Criminal Law and Philosophy* 2.2, S. 163–180. <https://doi.org/10.1007/s11572-007-9042-1>.
- (2008b): Legal and Technological Normativity. More (or Less) than Sisters. In: *Techné: Research in Philosophy and Technology* 12.3, S. 169–183.
- (2008c): Profiling and the Rule of Law. In: *Identity in the Information Society* 1.1, S. 55–70. <https://doi.org/10.1007/s12394-008-0003-1>.
- (2009): Technology and the End of Law. In: *Facing the Limits of the Law*. Hrsg. von Bert Keirsbilck; Wouter Devroe; Erik Claes. Berlin, Heidelberg: Springer, S. 1–22.
- Hill, Benjamin Mako u. a. (2006): *The Official Ubuntu Book*. New Jersey: Prentice Hall.
- Hodges, Andrew (1983): *Alan Turing: The Enigma*. New York: Simon and Schuster.
- Holtgrewe, Ursula; Raymund Werle (2001): De-Commodifying Software? Open Source Software between Business Strategy and Social Movement. In: *Science Studies* 14.2, S. 43–65.
- Iivari, Netta (2009a): ‘Constructing the Users’ in Open Source Software Development: An Interpretive Case Study of User Participation. In: *Information Technology & People* 22.2, S. 132–156.
- (2009b): Empowering the Users? A Critical Textual Analysis of the Role of Users in Open Source Software Development. In: *AI & Society* 23.4, S. 511–528. <https://doi.org/10.1007/s00146-008-0182-1>.
- Introna, Lucas D. (2007): Maintaining the Reversibility of Foldings: Making the Ethics (Politics) of Information Technology Visible. In: *Ethics and Information Technology* 9.1, S. 11–25. <https://doi.org/10.1007/s10676-006-9133-z>.
- Introna, Lucas D.; David Wood (2004): Picturing Algorithmic Surveillance: The Politics of Facial Recognition Systems. In: *Surveillance & Society* 2.2/3.
- Jensen, Carlos; Scott King; Victor Kuechler (2011): Joining Free/Open Source Software Communities: An Analysis of Newbies’ First Interactions on Project Mailing Lists. In: *System Sciences (HICSS) 2011, 44th Hawaii International Conference on*. IEEE, S. 1–10.
- Kitchin, Rob (2017): Thinking Critically about and Researching Algorithms. In: *Information, Communication & Society* 20.1, S. 14–29. <https://doi.org/10.1080/1369118X.2016.1154087>.
- Kittler, Friedrich (1993): Es gibt keine Software. In: *Draculas Vermächtnis: Technische Schriften*. Leipzig: Reclam, S. 225–242.

- Kline, Ronald; Trevor Pinch (1996): Users as Agents of Technological Change: The Social Construction of the Automobile in the Rural United States. In: *Technology and Culture* 37.4, S. 763–795.
- Knorr-Cetina, Karin (1999): *Epistemic Cultures: How the Sciences Make Knowledge*. Cambridge [u. a.]: Harvard University Press.
- Koops, Bert-Jaap (2007): Criteria for Normative Technology: An Essay on the Acceptability of ‘Code as Law’ in Light of Democratic and Constitutional Values. In: *Regulating Technologies. Legal Futures, Regulatory Frames and Technological Fixes*. Hrsg. von Roger Brownsword; Karen Yeung. Oxford, Portland: Hart Publishing, S. 157–174.
- Krogh, Georg von; Eric von Hippel (2006): The Promise of Research on Open Source Software. In: *Management Science* 52.7, S. 975–983. <https://doi.org/10.1287/mnsc.1060.0560>.
- Kuechler, Victor; Claire Gilbertson; Carlos Jensen (2012): Gender Differences in Early Free and Open Source Software Joining Process. In: *Open Source Systems: Long-Term Sustainability. 8th IFIP WG 2.13 International Conference, OSS 2012*. Berlin: Springer, S. 78–93.
- Kuhn, Thomas (1970): *The Structure of Scientific Revolutions*. Hrsg. von Otto Neurath. Chicago [u. a.]: The Univ. of Chicago Press.
- Kuk, George (2006): Strategic Interaction and Knowledge Sharing in the KDE Developer Mailing List. In: *Management Science* 52.7, S. 1031–1042.
- Kuschner, David (2012): Machine Politics. The Man who Started the Hacker Wars. In: *The New Yorker*. URL: <https://www.newyorker.com/magazine/2012/05/07/machine-politics> (besucht am 05.09.2018).
- Lakhani, Karim R.; Robert G. Wolf (2005): Why Hackers Do What They Do: Understanding Motivation and Effort in Free/Open Source Software Projects. In: *Perspectives on Free and Open Source Software*. Hrsg. von Joseph Feller u. a. Cambridge: MIT Press, S. 3–22.
- Latour, Bruno (1991): Technology Is Society Made Durable. In: *A Sociology of Monsters: Essays on Power, Technology and Domination*. Hrsg. von John Law. London: Routledge.
- (1992): Where Are the Missing Masses? The Sociology of a Few Mundane Artifacts. In: *Shaping Technology/Building Society*. Hrsg. von Wiebe E Bijker; John Law. Cambridge: MIT Press.
- Lazaro, Christophe (2008): *La liberté logicielle. Une ethnographie des pratiques d’échange et de coopération au sein de la communauté Debian*. Louvain-la-Neuve: Academia Bruylant.

- (2009). From Source Code to Text Code: Textual Norms and Practices in the Debian Community. URL: <http://http://constantvw.org/verlag/spip.php?article119> (besucht am 13.06.2013).
- Leenes, Ronald; Bert-Jaap Koops (2005): ‘Code’: Privacy’s Death or Saviour? In: *International Review of Law, Computers & Technology* 19.3, S. 329–340. <https://doi.org/10.1080/13600860500348572>.
- Lessig, Lawrence (1999): *Code and Other Laws of Cyberspace*. New York: Basic Books.
- Lyon, David (2003): Surveillance as Social Sorting. Computer Codes and Mobile Bodies. In: *Surveillance As Social Sorting: Privacy, Risk, and Digital Discrimination*. London, New York: Routledge, S. 13–30.
- Mackay, Hugh u. a. (2000): Reconfiguring the User: Using Rapid Application Development. In: *Social Studies of Science* 30.5, S. 737–757. <https://doi.org/10.1177/030631200030005004>.
- Marcovich, Anne; Terry Shinn (2012): Regimes of Science Production and Diffusion: Towards a Transverse Organization of Knowledge. In: *Scientiae Studia* 10 special edition São Paulo, S. 33–64.
- Marx, Gary T. (2002): What’s New About the “New Surveillance”? Classifying for Change and Continuity. In: *Surveillance & Society* 1.1, S. 9–29.
- Mateos-Garcia, Juan; W. Edward Steinmueller (2008): The Institutions of Open Source Software: Examining the Debian Community. In: *Information Economics and Policy* 20.4, S. 333–344.
- Morozov, Evgeny (2013): *To Save Everything, Click Here: The Folly of Technological Solutionism*. New York: PublicAffairs.
- Neff, Gina; Joshua McVeigh-Schultz; Tim Jordan (2012): Affordances, Technical Agency, and the Politics of Technologies of Cultural Production. In: *Culture Digitally*. URL: <http://culturedigitally.org/2012/01/affordances-technical-agency-and-the-politics-of-technologies-of-cultural-production-2/> (besucht am 03.10.2018).
- O’Mahony, Siobhán (2007): The Governance of Open Source Initiatives: What Does It Mean to Be Community Managed? In: *Journal of Management & Governance* 11.2, S. 139–150.
- O’Mahony, Siobhán; Fabrizio Ferraro (2007): The Emergence of Governance in an Open Source Community. In: *Academy of Management Journal* 50.5, S. 1079 bis 1106.
- Pariser, Eli (2011): *The Filter Bubble: What the Internet Is Hiding from You*. New York [u. a.]: Penguin Press.

- Perrow, Charles (1987): *Normale Katastrophen. Die unvermeidlichen Risiken der Großtechnik*. Frankfurt/M. [u.a.]: Campus.
- Pestre, Dominique (2003): Regimes of Knowledge Production in Society: Towards a More Political and Social Reading. In: *Minerva* 41.3, S. 245–261.
- Phillips, P. J. u. a. (2003): Face Recognition Vendor Test (FRVT) 2002: Overview and Summary. Techn. Ber. URL: <https://www.nist.gov/document/frvt2002overviewandsummarypdf> (besucht am 05.03.2020).
- Rajanen, Mikko; Netta Iivari; Arto Lanamäki (2015): Non-Response, Social Exclusion, and False Acceptance: Gatekeeping Tactics and Usability Work in Free-Libre Open Source Software Development. In: *Human-Computer Interaction – INTERACT 2015*. Cham: Springer International Publishing, S. 9–26.
- Rammert, Werner (2004): Two Styles of Knowing and Knowledge Regimes: Between ‘Explicitation’ and ‘Exploration’ under Conditions of ‘Functional Specialization’ or ‘Fragmental Distribution’. TUTS Working Papers. URL: https://www.ts.tu-berlin.de/fileadmin/fg226/TUTS/TUTS_WP_3_2004.pdf (besucht am 12.04.2020).
- (2006): Technik in Aktion: Verteiltes Handeln in soziotechnischen Konstellationen. In: *Technografie. Zur Mikrosoziologie der Technik*. Frankfurt/M.: Campus, S. 163–195.
- Robles, Gregorio; Laura Arjona Reina u. a. (2016): Women in Free/Libre/Open Source Software: The Situation in the 2010s. In: *Open Source Systems: Integrating Communities*. Hrsg. von Kevin Crowston u. a. Cham: Springer International Publishing, S. 163–173. https://doi.org/10.1007/978-3-319-39225-7_13.
- Robles, Gregorio; Hendrik Scheider u. a. (2001): Who Is Doing It. A research on Libre Software developers. URL: <https://pdfs.semanticscholar.org/1a3f/3184edc33a93458ca39dcc3e12d941916700.pdf> (besucht am 03.10.2018).
- Sack, Warren u. a. (2006): A Methodological Framework for Socio-Cognitive Analyses of Collaborative Design of Open Source Software. In: *Computer Supported Cooperative Work (CSCW)* 15.2–3, S. 229–250.
- Schäufele, Fabia (2017): *Profiling zwischen sozialer Praxis und technischer Prägung. Ein Vergleich von Flughafensicherheit und Credit-Scoring*. Wiesbaden: Springer VS Verlag für Sozialwissenschaften.
- Schräpe, Jan-Felix (2016): *Open-Source-Projekte als Utopie, Methode und Innovationsstrategie: Historische Entwicklung – Soziökonomische Kontexte – Typologie*. Glückstadt: Hülsbusch.
- Schulz-Schaeffer, Ingo (2000): *Sozialtheorie der Technik*. Frankfurt/M., New York: Campus [Zugl.: Diss., Univ. Bielefeld 1999].

- Schützeichel, Rainer (2010): Wissen, Handeln, Können. German. In: *Soziologie der Kompetenz*. Hrsg. von Thomas Kurtz; Michaela Pfadenhauer. Wiesbaden: VS Verlag für Sozialwissenschaften, S. 173–189.
- (2012): Wissenssoziologie. In: *Handbuch Wissenschaftssoziologie*. Hrsg. von Sabine Maasen u.a. Wiesbaden: Springer Fachmedien, S. 17–26.
- Sebald, Gerd (2008): *Offene Wissensökonomie. Analysen zur Wissenssoziologie der Free/Open Source-Softwareentwicklung*. Wiesbaden: VS Verlag für Sozialwissenschaften.
- Shah, Sonali K. (2006): Motivation, Governance, and the Viability of Hybrid Forms in Open Source Software Development. In: *Management Science* 52.7, S. 1000 bis 1014. <https://doi.org/10.1287/mnsc.1060.0553>.
- Sowe, Sulayman; Ioannis Stamelos; Lefteris Angelis (2006): Identifying Knowledge Brokers That Yield Software Engineering Knowledge in OSS Projects. In: *Information and Software Technology* 48.11, S. 1025–1033.
- Spiegel, André (2006): *Die Befreiung der Information. GNU, Linux und die Folgen*. Berlin: Matthes & Seitz.
- Stallman, Richard Matthew (1986): What Is the Free Software Foundation? In: *GNU's Bulletin* 1.1, S. 8–9. URL: <https://www.gnu.org/bulletins/bull1.txt> (besucht am 03.10.2018).
- (2002): *Free Software, Free Society. Selected Essays of Richard M. Stallman*. Hrsg. von Joshua Gay. Boston: GNU Press.
- (2013): Why Free Software Is More Important Now Than Ever Before. In: *Wired*. URL: <https://www.wired.com/2013/09/why-free-software-is-more-important-now-than-ever-before/> (besucht am 01.10.2018).
- Stegbauer, Christian (2009): *Wikipedia. Das Rätsel der Kooperation*. Wiesbaden: VS Verlag für Sozialwissenschaften.
- Taubert, Niels C. (2006): *Produktive Anarchie? Netzwerke Freier Softwareentwicklung*. Bielefeld: transcript.
- Turkle, Sherry (1995): *Life on the Screen*. New York [u.a.]: Simon and Schuster.
- Van Oost, Ellen (2003): Materialized Gender: How Shavers Configure the Users' Femininity and Masculinity. In: *How Users Matter*. Hrsg. von Nelly Oudshoorn; Trevor Pinch. Cambridge: MIT Press, S. 193–208.
- Vasilescu, Bogdan u. a. (2015): Gender and Tenure Diversity in GitHub Teams. In: *CHI '15: Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*. New York: ACM Press, S. 3789–3798. <https://doi.org/10.1145/2702123.2702549>.

- Wallace, James; James Erickson (1993): *Hard Drive: Bill Gates and the Making of the Microsoft Empire*. New York: Harper Business.
- Wehling, Peter (2007): Wissensregime. In: *Handbuch Wissenssoziologie und Wissensforschung*. Hrsg. von Rainer Schützeichel. Konstanz: UVK, S. 704–712.
- Weick, Karl E. (1990): Technology as Equivoque. Sensemaking in New Technologies. In: *Technology and Organizations*. Hrsg. von Paul S. Goodman; Lee S. Sproull; Associates. San Francisco [u.a.]: Jossey-Bass Publishers, S. 1–44.
- Wenger, Etienne (1998): Communities of Practice: Learning as a Social System. In: *Systems Thinker* 9.5, S. 2–3.
- West, Joel; Siobhán O'Mahony (2005): Contrasting Community Building in Sponsored and Community Founded Open Source Projects. In: *System Sciences, 2005. HICSS'05. Proceedings of the 38th Annual Hawaii International Conference on. IEEE*, S. 196c.
- (2008): The Role of Participation Architecture in Growing Sponsored Open Source Communities. In: *Industry and Innovation* 15.2, S. 145–168.
- Winner, Langdon (1980): Do Artifacts Have Politics? In: *Daedalus* 109.1, S. 121–136.
- Wood, David; Stephen Graham (2006): Permeable Boundaries in the SoftwareSorted Society: Surveillance and the Differentiation of Mobility. In: *Mobile Technologies of the City*. Hrsg. von Mimi Sheller u. a. London: Routledge, S. 177–191.
- Woolgar, Steve (1991): Configuring the User. The Case of Usability Trials. In: *A Sociology of Monsters: Essays on Power, Technology and Domination*. Hrsg. von John Law. London: Routledge, S. 58–99.
- Young, Michael Dunlop (1958): *The Rise of the Meritocracy*. New Brunswick, London: Transaction Publishers.
- Ziewitz, Malte (2016): Governing Algorithms: Myth, Mess, and Methods. In: *Science, Technology, & Human Values* 41.1, S. 3–16. <https://doi.org/10.1177/0162243915608948>.
- Zittel, Claus (2014): Wissenskulturen, Wissensgeschichte und historische Epistemologie. In: *Rivista Internazionale di Filosofia e Psicologia* 5.1, S. 29–42.

Weitere Titel aus dem vwh-Verlag (Auszug)

- S. Munz, J. Soergel: Agile Produktentwicklung im Web 2.0 2007, ~~32,90 €~~
UVP: 13,80 €, ISBN 978-3-940317-11-7
- J. Moskaliuk (Hg.): Konstruktion und Kommunikation von Wissen mit Wikis 2008, 27,50 €, ISBN 978-3-940317-29-2
- G. Vogl: Selbstständige Medienschaffende in der Netzwerkgesellschaft 2008 ~~29,90 €~~ UVP: 8,50 €, ISBN 978-3-940317-38-4
- S. Mühlbacher: Information Literacy in Enterprises 2009, ~~32,90 €~~ UVP: 9,80 €, ISBN: 978-3-940317-45-4
- M. Maßun: Collaborative Information Management in Enterprises 2009, ~~28,90 €~~ UVP: 8,90 €, ISBN 978-3-940317-49-0
- J. Griesbaum: Mehrwerte des kollaborativen Wissensmanagements in der Hochschullehre 2009, ~~35,90 €~~ UVP: 7,80 € ISBN 978-3-940317-52-0
- T. Memmel: User Interface Specification for Interactive Software Systems 2009 ~~33,90 €~~ UVP: 9,80 €, ISBN 978-3-940317-53-7
- A. Ratzka: Patternbasiertes User Interface Design für multimodale Interaktion 2010, ~~33,90 €~~ UVP: 10,80 €, 978-3-940317-62-9
- M. Janneck, C. Adelberger: Komplexe Software-Einführungsprozesse gestalten: Grundlagen und Methoden Am Beispiel eines Campus-Management-Systems 2012, 26,90 €, 978-3-940317-63-6
- C. Russ: Online Crowds Massenphänomene und kollektives Verhalten im Internet 2010, 31,50 €, 978-3-940317-67-4
- M. Prestipino: Die virtuelle Gemeinschaft als Informationssystem Informationsqualität nutzergenerierter Inhalte in der Domäne Tourismus 2010, ~~30,90 €~~ UVP: 9,80 €, ISBN 978-3-940317-69-8
- A. Warta: Kollaboratives Wissensmanagement in Unternehmen Indikatoren für Erfolg und Akzeptanz am Beispiel von Wikis 2011, 30,90 €, ISBN 978-3-940317-90-2
- M. Görtz: Social Software as a Source of Information in the Workplace 2011, 31,90 €, ISBN 978-3-86488-006-3
- H. Fritzlar, A. Huber, A. Rudl (Hg.): Open Source im Public Sector: günstiger, sicherer, flexibler 2012, ~~25,90 €~~ UVP: 5,90 €, ISBN 978-3-86488-013-1
- J.-F. Schrape: Wiederkehrende Erwartungen Visionen, Prognosen und Mythen um neue Medien seit 1970 2012, 11,90 €, ISBN 978-3-86488-021-6
- S.-J. Untiet-Kepp: Adaptive Feedback zur Unterstützung in kollaborativen Lernumgebungen 2012, 30,90 €, ISBN 978-3-86488-023-0
- A. Hiller, M. Schneider, A. C. Wagner: Social Collaboration Workplace Das neue Intranet erfolgreich einführen 2014, 26,90 €, ISBN 978-3-86488-065-0
- B. Heuwing: Usability-Ergebnisse als Wissensressource in Organisationen 2015, 35,80 €, ISBN 978-3-86488-084-1
- U. Herb: Open Science in der Soziologie Eine interdisziplinäre Bestandsaufnahme zur offenen Wissenschaft und eine Untersuchung ihrer Verbreitung in der Soziologie 2015, 36,80 €, 978-3-86488-083-4
- J.-F. Schrape: Open-Source-Projekte als Utopie, Methode und Innovationsstrategie Historische Entwicklung – sozio-ökonomische Kontexte – Typologie 2016, 17,90 €, ISBN 978-3-86488-089-6
- M. Kainz: Globale Vernetzung – globale Identität? Kulturelle Identitätskonstruktionen im Zeitalter digitaler Technologien 2018, 33,80 €, ISBN 978-3-86488-131-2
- K. Kessler: Die (Ohn-)Macht des Prosumenten Eine kritische Betrachtung des neuen Rollenverhältnisses von Konsument u. Produzent 2018, 32,80 €, 978-3-86488-135-0
- T. Veigl: Machinima Kultur, Ästhetik und Ökonomie einer digitalen User Innovation 2019, 37,80 €, ISBN 978-3-86488-157-2



Aktuelle Ankündigungen, Inhaltsverzeichnisse und Rezensionen finden Sie im vwh-Blog unter www.vwh-verlag.de.

Das komplette Verlagsprogramm mit Buchbeschreibungen sowie eine direkte Bestellmöglichkeit im vwh-Shop finden Sie unter www.vwh-verlag-shop.de.