Ökonomische Analyse offener Software

Inauguraldissertation zur Erlangung des akademischen Grades eines
Doktors der Wirtschaftswissenschaften der Universität Mannheim

Jörg Gutsche

vorgelegt im
Wintersemester 2005/2006

Referent: Prof. Dr. Klaus Conrad

Koreferent: Prof. Dr. Siegfried Berninghaus (Universität Karlsruhe (TH))

Dekan: Prof. Konrad Stahl, Ph.D.

Tag der mündlichen Prüfung: 9. Februar 2006

Danksagung

Die nachfolgend aufgeführten Menschen haben in verschiedener Weise zum Gelingen der vorliegenden Arbeit beigetragen. Ihnen allen bin ich zu Dank verpflichtet.

Insbesondere danke ich meinem Doktorvater Herrn Professsor Dr. Klaus Conrad für seine Bereitschaft, mich in dem von mir gewählten Thema zu betreuen, seinen wissenschaftlichen Rat und nicht zuletzt seine wohlwollende, zutiefst menschliche Art.

Herrn Professor Dr. Siegfried Berninghaus danke ich für die uneigennützige Übernahme des Koreferats, den Herren Professoren Dr. Paul Gans und Felix Kübler, Ph.D. für ihre Beteiligung am Prüfungsausschuß.

Meinen Lehrstuhl-Kollegen Peter Hasfeld, Gerrit Löber, Yvonne Reiter, Matthias Staat und Max danke ich für die kooperative und freundschaftliche Zusammenarbeit. Es war eine schöne Zeit mit euch.

Und ganz besonders herzlich möchte ich mich bei meinen Eltern sowie meinen Geschwistern Jens und Katja bedanken. Ohne ihren Rückhalt wäre diese Arbeit vermutlich nicht geglückt.

Mannheim, den 13. Februar 2006

Jörg Gutsche

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Abkürzungsverzeichnis

ACS	Affiliated Computer Services
API	Application Programming Interface
ASF	Apache Software Foundation
AT&T	American Telephone & Telegraph Company
BIND	Berkeley Internet Name Domain
BSD	Berkeley Software Distribution
CDU	Christlich Demokratische Union
COTS	commercial-off-the-shelf
CSC	Computer Sciences Corporation
CVS	Concurrent Versions System
EAL2	Evaluation Assurance Level 2
EDS	Electronic Data Systems Corporation
FDP	Freie Demokratische Partei
FSF	Free Software Foundation
FUD	Fear, Uncertainty, and Doubt
GNOME	GNU Object Model Environment
GNU	GNU's not Unix
GPL	General Public License
IBM	International Business Machines
IDC	International Data Corporation
IEE	Internet Information Server
IT	Informationstechnologie
KDE	K Desktop Environment
LGPL	Lesser General Public License

LKA	Linux Kernel Archive
LKML	Linux Kernel Mailing List
MIT	Massachusetts Institute of Technology
MPL	Mozilla Public License
NASDAQ	National Association of Securities Dealers Automated Quotation System
NCSA	National Center for Supercomputing Applications
NPL	Netscape Public License
NTT	Nippon Telegraph and Telephone Corporation
OECD	Organisation for Economic Co-operation and Development
OSD	Open Source Definition
OSI	Open Source Initiative
OSS	Open Source Software
PDS	Partei des Demokratischen Sozialismus
PHP	PHP Hypertext Preprocessor
SAP	Systeme, Anwendungen, Produkte
SPD	Sozialdemokratische Partei Deutschlands
SQL	Structured Query Language
TCO	Total Cost of Ownership
UNESCO	United Nations Educational, Scientific and Cultural Organization
USA	United States of America

1 Einleitung

1.1 Motivation

In den letzten Jahren wurde die Softwareindustrie von einem neuartigen Phänomen, sogenannter offener Software, nachhaltig erschüttert. Das konstituierende Charakteristikum dieser Art von Software ist, daß die sie begleitende Lizenz ihren Benutzern deutlich weitreichendere Rechte einräumt, als dieses üblicherweise bei proprietären Lizenzen der Fall ist. Insbesondere ist es erlaubt, den Quelltext der Software, der als ihre menschgerechte Repräsentation und somit als Träger ihrer Architektur und der in ihr enthaltenen Ideen und Technologien gelten kann, einsehen, modifizieren und ohne die Zahlung von Lizenzgebühren unter der gleichen Lizenz weitergeben zu dürfen.

Als Folge dieser Lizenzgestaltung trägt der Vorgang der eigentlichen Entwicklung offener Software äußerst eigenwillige Züge: In der Regel arbeiten in einem offenen Softwareprojekt zahlreiche unabhängige Programmierer – als Privatpersonen, Angestellte eines Unternehmens oder als Wissenschaftler – aus aller Welt parallel an unterschiedlichen Aspekten des Programms. Sie koordinieren dabei ihre Anstrengungen in einem scheinbar anarchischen Prozeß über die vom Internet bereitgestellten Kommunikationsmöglichkeiten und erhalten für ihre Arbeit keine unmittelbare monetäre Kompensation. In Anlehnung an das turbulente Treiben auf einem orientalischen Basar bezeichnet Raymond (2001) dieses Vorgehen als die Basar-Methode der Softwareentwicklung, die er dem traditionellen Ansatz gegenüberstellt, bei dem die Programmierer unter der Aufsicht eines Baumeisters nach den strengen Vorgaben eines umfassenden Plans vorgehen, und der von ihm deshalb Kathedralen-Methode genannt wird.

Die wichtigste unmittelbare ökonomische Implikation der oben beschriebenen Lizenzgestaltung besteht darin, daß offene Software in der Regel kostenlos erhältlich ist: Aufgrund der genannten Rechte kann jeder als Anbieter einer offenen Software in den Markt eintreten, ohne auch nur einen Teil der Entwicklungskosten tragen zu müssen. Des weiteren steht mit dem Internet eine hocheffiziente Infrastruktur zum Austausch digitaler Informationen zur Verfügung; de facto sind die fixen und variablen Kosten, die beim Vertrieb von Software über das Internet anfallen, vernachlässigbar gering. Selbstverständlich kann sich in einer solchen Situation nur ein Gleichgewichtspreis in Höhe von Null etablieren.

In zahlreichen Märkten ist offene Software zu einem ernstzunehmenden Konkurrenten proprietärer Angebote herangewachsen. Prominente offene Softwarepakete wie das Betriebssystem GNU/Linux, der Web-Server Apache oder die E-Mail-Verteilungssoftware Sendmail sind in einigen Teilmärkten sogar unangefochtene Marktführer.

Dieses ist nicht nur auf den offensichtlichen Lizenzkostenvorteil gegenüber proprietärer Software zurückzuführen, der – wie zahlreiche Studien und Beispielkalkulationen belegen – für viele Nutzungsszenarien auch dann bestehen bleibt, wenn man anstelle eines reinen Lizenzgebührenvergleichs eine umfassende Analyse aller mit dem Einsatz einer Software verbundenen Kosten, etwa für Inbetriebnahme, Schulung und Wartung, vornimmt. Empirische und auch theoretische Untersuchungen zeigen, daß offene Software auch anderen entscheidungsrelevanten Qualitätskriterien wie etwa Funktionsvielfalt, Fehlerfreiheit, Sicherheit, Geschwindigkeit, Robustheit, Skalierbarkeit und Benutzerfreundlichkeit durchaus gerecht wird.

Zahlreiche etablierte IT-Unternehmen haben inzwischen Teile ihrer Geschäftsstrategie an offene Software angepaßt. So investieren Branchengrößen wie IBM, Sun und Hewlett-Packard signifikante Summen in die Weiterentwicklung bestehender offener Software und haben Hardwarelinien eingeführt, die dezidiert für den Einsatz unter Linux konzipiert sind. Das Softwareunternehmen Netscape, inzwischen Teil des Medienkonzerns Time Warner, veröffentlichte im Jahr 1998 seinen Web Browser Netscape Communicator, bis dahin eine proprietäre

Software, unter einer offenen Lizenz; ähnliches unternahm Sun im Jahr 2000 mit StarOffice, einem Paket von Büroanwendungen. Microsoft sieht offene Software inzwischen als große Bedrohung für den langfristigen Erfolg der eigenen Strategie an und reagiert mit einer Reihe von Gegenmaßnahmen.

Zudem sind in den letzten Jahren zahlreiche neue Unternehmen wie etwa Red Hat und MySQL gegründet worden, in deren ureigenen Geschäftskonzepten offene Software eine zentrale Rolle einnimmt. Und selbst die öffentliche Hand und überstaatliche Organisationen sind inzwischen auf offene Software aufmerksam geworden. In teilweise hitzigen Debatten wird diskutiert, inwieweit der Staat offene Software fördern sollte.

1.2 Forschungsrichtungen

Offensichtlich hat offene Software in der gesamten IT-Branche und ihrem Umfeld bereits jetzt tiefgreifende Veränderungen bewirkt; einige Analysten sprechen gar von einem disruptiven Phänomen, welches sämtliche Branchenspielregeln zu redefinieren vermag. Es ist daher geboten, die ökonomischen Implikationen offener Software genau zu verstehen. Dabei lassen sich die folgenden ökonomischen Forschungsrichtungen abgrenzen:

- Die ersten Arbeiten befassen sich mit der Frage, welche Motive in Ermangelung von Lizenzgebühren die Entwicklung offener Software bewirken. Repräsentative Arbeiten dieser Richtung stammen von Gosh et al. (2002), Lerner und Tirole (2002) sowie Lakhani und Wolf (2005).

- Darauf aufbauend analysiert eine gut entwickelte theoretische Literatur die in der Basar-Methode der Softwareentwicklung angelegten Koordinationsprobleme. Beispielhaft sind die Arbeiten von Johnson (2002), Myatt und Wallace (2002) sowie Bitzer und Schröder (2005).

- Verschiedene industrieökonomisch geprägte Arbeiten analysieren die Auswirkungen offener Software auf Märkte und leiten daraus Gründe für

staatliches Eingreifen zugunsten oder gegen offene Software ab. Typische Vertreter dieser Literatur sind Schmidt und Schnitzer (2003), Comino und Manenti (2005) und Mustonen (2005).

- Schließlich wird versucht, aus der Basar-Methode Einsichten über neue Organisationsstrukturen und kollektive Innovationsprozesse zu gewinnen. Repräsentative Arbeiten dieser Richtung haben Harhoff, Henkel und von Hippel (2003), von Hippel und von Krogh (2003) und Henkel (2004a) vorgelegt.

Diese Arbeit erweitert die bestehende Literatur zur Koordinationsproblematik und die zur Frage der Notwendigkeit staatlichen Handelns.

1.3 Aufbau dieser Arbeit

Zunächst wird in Kapitel 2 das Phänomen "Offene Software" in all jenen Facetten, die für den Gang dieser Untersuchung relevant sind, vorgestellt. Auf die eingehende Beschreibung der eingangs bereits erwähnten Lizenzgestaltung und ihrer Implikationen folgen ein kurzer Abriß der Entstehungsgeschichte offener Software und eine genaue Beschreibung der Strukturen, in denen diese Software gegenwärtig entwickelt und genutzt wird.

Kapitel 3 behandelt die Entwicklung offener Software. Nach einer kurzen Übersicht über die empirisch relevanten Motive für die Beteiligung an offenen Softwareprojekten werden einige ausgewählte Modelle zur Koordination vorgestellt. Anschließend wird ein Modell dazu entwickelt, welche Bedingungen die Entstehung offener Software fördern. Dieses steht im Kontrast zur bisherigen Forschung, die hauptsächlich analysiert, wie und warum bereits als Institution etablierte offene Softwareprojekte funktionieren, aber kaum erklären kann, wie derartige Projekte "aus dem Nichts" entstehen. Relevant ist dieser Wechsel der Perspektive in zweifacher Hinsicht: Zum einen erlaubt er Aussagen darüber, in welchen Bereichen noch mit der Entstehung offener Software zu rechnen ist.

Zum anderen ermöglicht er aber auch Prognosen darüber, welche derzeit existierenden offenen Softwareprojekte bleiben und welche sich als Modeerscheinung ohne Dauer erweisen werden.

In Kapitel 4 wird im wesentlichen untersucht, ob ein Bedarf für staatliche Eingriffe in Märkte für Massensoftware zugunsten offener Software besteht. Anders als in der bestehenden Literatur werden dabei in einem einheitlichen Modellrahmen systematisch alle Spezifika von Softwaremärkten berücksichtigt. Anschließend wird zudem die Bedeutung offener Software in Märkten für Individualsoftware diskutiert.

Schließlich werden in Kapitel 5 die Hauptergebnisse dieser Arbeit zusammengefaßt und in ihrer Gesamtheit kritisch gewürdigt.

2 Grundlagen zu offener Software

Ziel dieses Kapitels ist es, alle für das Verständnis der weiteren Betrachtungen erforderlichen Grundlagen zu offener Software darzulegen. Dazu wird zunächst formal definiert, was unter offener Software zu verstehen ist; anschließend wird diese formale Definition über eine Darstellung der besonderen Kultur, die das Konzept offener Software hervorgebracht hat, verbreitert und abgerundet. Es folgt eine Beschreibung aller Gruppen und Institutionen, welche im Umfeld offener Software wesentliche Aufgaben wahrnehmen. Eine Diskussion der Spezifika offener Software aus Entwickler- sowie aus Nutzersicht beschließt das Kapitel.

2.1 Begriffsdefinition

2.1.1 Offene Lizenzen

Das Konzept offener Software ist, obgleich gewachsen, klar definiert. Zu verdanken ist dieses den intensiven Bemühungen der Programmierer-Kultur, welche die Idee hervorgebracht hat, schon frühzeitig die wesentlichen Merkmale offener Software herauszuarbeiten. Als maßgeblich gilt heutzutage der Merkmalskatalog, welchen die gemeinnützige *Open Source Initiative* (OSI), aufbauend auf der *Free Software Definition* und den *Debian Free Software Guidelines*,[1] im Jahr 1998 mit der sogenannten *Open Source Definition* (OSD) vorgelegt hat. Demnach gilt eine Software genau dann als offen, wenn die Lizenz, unter der diese Software verbreitet wird, die folgenden vier Eigenschaften aufweist:

[1] Weitere Informationen zur Free Software Definition und zu den Debian Free Software Guidelines finden sich unter `www.fsf.org/philosophy/free-sw.html` und `www.debian.org/social_contract.hmtl`.

1. **Uneingeschränkte Weiterverbreitung** Die Lizenz muß die uneingeschränkte Weiterverbreitung der Software gestatten. Insbesondere dürfen hierfür keine Lizenzgebühren erhoben werden.

2. **Keine Diskriminierung** Die Lizenz darf keine Bestimmungen enthalten, die abhängig vom Nutzer, dem Verwendungszweck oder dem technologischen Einsatzumfeld sind.

3. **Verfügbarkeit des Quelltextes** Die Lizenz muß vorschreiben, daß die Software prinzipiell immer gemeinsam mit dem Quelltext, der als ihre menschgerechte Repräsentation und somit als Träger ihrer Architektur und der in ihr enthaltenen Ideen und Technologien angesehen werden kann, weitergegeben wird. Auf die gemeinsame Verbreitung von Software und Quelltext darf nur dann verzichtet werden, wenn die Möglichkeit besteht, den Quelltext bei Interesse leicht zu beschaffen. Des weiteren muß der Quelltext in einer Form vorliegen, welche seine Nutzung durch Menschen nicht behindert.

4. **Modifizierbarkeit des Quelltextes** Schließlich muß es die Lizenz erlauben, den Quelltext beliebig zu verändern und die so entstehende neue Software wiederum unter derselben Lizenz zu verbreiten.

Zudem beinhaltet die OSD weitere Anforderungen minderer Bedeutung, die dazu dienen, diverse rechtliche Schlupflöcher zu schließen.[2] Das Wesen offener Software wird aber durch die oben angegebenen vier Eigenschaften bereits genau erfaßt. Die Eigenschaften 1 und 2 charakterisieren offene Software aus der Sicht eines einfachen Nutzers, welcher das Recht erhält, die Software lizenzgebührenfrei in beliebiger Weise zu nutzen. Die Eigenschaften 3 und 4 charakterisieren offene Software hingegen aus der Sicht eines Softwareentwicklers, welcher das Recht erhält, die Software ohne Einschränkungen zu studieren und weiterzuentwickeln. Insbesondere dieses Recht auf freie Weiterentwicklung ist

[2]Der interessierte Leser sei auf Anhang A verwiesen, der die OSD im Volltext enthält.

ein zentrales Anliegen der Verfechter offener Software.[3]

Offensichtlich unterscheiden sich die Rechte, welche offene Lizenzen den Nutzern einer Software geben, ganz wesentlich von den durch proprietäre Lizenzen eingeräumten. Diese verbieten im Regelfall die lizenzgebührenfreie Weiterverbreitung der Software und enthalten diskriminierende Elemente (z.B. Studentenlizenzen). Des weiteren wird der Quelltext proprietärer Software üblicherweise als wichtiges Geschäftsgeheimnis angesehen und dementsprechend nicht veröffentlicht; auch ist die Modifikation proprietärer Software meistens untersagt. Es existiert aber auch proprietäre Software, welche zumindest einige der oben angegebenen Eigenschaften offener Software aufweist. Insofern sind die vier angegebenen Anforderungen tatsächlich alle notwendig.

Obgleich die OSD die grundlegenden Eigenschaften offener Software sehr klar umreißt, bietet sie dennoch einen gewissen Gestaltungsspielraum. Innerhalb dieses Spielraums existiert eine Vielzahl verschiedener offener Lizenzen, die den Zielen und Interessen ihrer Autoren in besonderer Weise Rechnung tragen. Nachfolgend werden die maßgeblichen offenen Lizenzen und ihre Charakteristika vorgestellt:

Die beiden ältesten offenen Lizenzen wurden in den 80er Jahren von der gemeinnützigen *Free Software Foundation* (FSF) vorgelegt: die *General Public License* (GPL) und die *Lesser General Public License* (LGPL).[4] Beide Lizenzen stehen in besonders starkem Gegensatz zu proprietären. Daher werden sie üblicherweise als Copyleft-Lizenzen bezeichnet, wobei der Begriff "Copyleft" auf ein Wortspiel zurückgeht, welches den gebräuchlichen rechtlichen Hinweis "Copyright – All rights reserved" zu "Copyleft – All rights reversed" verdreht. Ein zentrales Element beider Lizenzen ist die Auflage, im Falle der Weiterverbreitung sämtliche Modifikationen am Quelltext wiederum unter der GPL bzw. der LGPL zugänglich machen zu müssen. Es ist also nicht möglich, ei-

[3] Eine umfassende Darstellung der Ziele, welche die OSD verfolgt, gibt Perens (1999), ihr geistiger Vater.

[4] Ausführliche Lizenztexte finden sich unter `www.fsf.org/licenses/gpl.html` bzw. `www.fsf.org/licenses/lgpl.html`.

ne Copyleft-Software zur Grundlage eines proprietären Produkts zu machen; sie und alle ihre Weiterentwicklungen bleiben für immer offen. Des weiteren darf Copyleft-Quelltext nicht mit Non-Copyleft-Quelltext kombiniert und in ausführbaren Binärkode übersetzt werden. Copyleft-Software ist somit sehr systematisch von anders lizenzierter Software abgeschottet. Die LGPL wurde ausdrücklich mit dem Ziel entworfen, diese Abschottung kontrolliert zu lockern. Sie erlaubt anders als die GPL die Verbindung der lizenzierten Software mit Non-Copyleft-Software auf binärer Ebene (Verlinkung). Im wesentlichen erlaubt diese Lockerung, auf einem GPL-lizenzierten Betriebssystem (z.B. Linux) ein proprietäres Anwendungsprogramm auszuführen, indem beide lediglich über eine LGPL-lizenzierte Zwischenschicht interagieren. Aus Nutzersicht kann also ein GPL-lizenziertes Betriebssystem wie ein proprietäres verwendet werden.

Wichtige Non-Copyleft-Lizenzen sind die an der Universität Berkeley entworfene *Berkeley Software Distribution* (BSD) sowie die vom Softwareunternehmen Netscape entwickelten *Mozilla Public License* (MPL) und *Netscape Public License* (NPL).[5] Insbesondere die BSD erlegt dem Lizenznehmer sehr viel weniger Pflichten auf als Copyleft-Lizenzen. So darf BSD-lizenzierte Software beliebig mit anderer Software kombiniert und auch modifiziert und unter einer beliebigen Lizenz weiterverbreitet werden. Die BSD kann also als sehr liberale offene Lizenz bezeichnet werden. Die MPL beschreitet hingegen einen Mittelweg zwischen der GPL und der BSD. Einerseits erlaubt sie wie die BSD die beliebige Verbindung mit anderer Software; andererseits müssen Modifikationen wie bei der GPL offen gelegt werden. Die NPL schließlich ist der Archetyp einer Lizenz, welche dem ursprünglichen Inhaber der Eigentumsrechte an der Software, in diesem Fall dem Unternehmen Netscape, exklusive Sonderrechte hinsichtlich der Modifikationen des Quelltextes durch andere einräumt.[6] Ansonsten entspricht sie der MPL. Insgesamt also sind Non-Copyleft-Lizenzen der proprietären Verwertung offener

[5] Genaue Lizenztexte finden sich unter `www.opensource.org/licenses/bsd-license.php`, `www.mozilla.org/MPL/MPL-1.1.html` bzw. `www.mozilla.org/MPL/NPL-1.1.html`.

[6] Aufgrund dieser Sonderrechte hat sich inzwischen die Rechtsauffassung durchgesetzt, die NPL sei keine offene Lizenz. Sie wird hier wegen ihrer historischen Bedeutung angeführt.

	GPL	LGPL	BSD	MPL	NPL
Verbindung auf binärer Ebene nur mit Copyleft-Software erlaubt	√				
Verbindung auf Quelltextebene nur mit Copyleft-Software erlaubt	√	√			
Modifikationen des Quelltextes müssen zugänglich gemacht werden	√	√		√	√
Sonderrechte für den ursprünglichen Inhaber der Eigentumsrechte					√

Tabelle 2.1: Eigenschaften wichtiger offener Lizenzen

Software deutlich zugeneigter als Copyleft-Lizenzen; BSD-lizenzierte Software darf sogar in jedweder Weise in ein proprietäres Produkt integriert werden.

Die wesentlichen Eigenschaften aller hier beschriebenen Lizenzen sind in Tabelle 2.1 zusammengestellt. Eine ausführliche Diskussion der Lizenzen mit besonderem Blick auf das deutsche Recht geben Jaeger und Metzger (2002).

Es ist schwierig, die empirische Bedeutung der hier genannten Lizenzen genau zu erfassen. Einen Anhaltspunkt gibt Misanec (2001), der untersucht, mit welcher Häufigkeit die verschiedenen Lizenzen in "SuSE Linux 7.2 Professional", einer beliebten Zusammenstellung von Software rund um das offene Betriebssystem Linux, auftreten. Es zeigt sich, daß die GPL am häufigsten verwendet wird; die LGPL und die BSD folgen mit weitem Abstand auf den Plätzen 2 und 3. Ein ähnliches Bild erbringt die Untersuchung der Lizenzverteilung auf SourceForge.net.[7] Die genauen Zahlen sind in Tabelle 2.2 zusammengestellt. Es sei aber angemerkt, daß die meisten hier nicht angegebenen Lizenzen eher der

[7]Die Internetseite SourceForge.net stellt den Entwicklern offener Software die für deren Arbeit erforderliche Infrastruktur zur Verfügung. Sie wird vom Unternehmen VA Software als Technologiedemonstration kostenfrei angeboten. Derzeit nutzen über 100.000 offene Entwicklungsprojekte diese Infrastruktur.

	GPL	LGPL	BSD	MPL
SuSE Linux 7.2 Professional	48 %	12 %	7 %	k.A.
SourceForge.net (Januar 2005)	69 %	11 %	7 %	2 %

Tabelle 2.2: Verwendungshäufigkeit wichtiger offener Lizenzen

BSD als der GPL entsprechen.[8] Daher kann man davon ausgehen, daß die an 100 % fehlenden Lizenzen fast alle BSD-artiger Natur sind.

2.1.2 Abgrenzung von anderen Lizenzierungsarten

Es existieren einige weitere Arten der Lizenzierung von Software, welche aufgrund gewisser Ähnlichkeiten häufig mit offenen Lizenzen verwechselt werden. Nachfolgend werden diese kurz dargestellt, um das Konzept offener Software noch klarer herauszuarbeiten.

1. **Public Domain Software** Sogenannte *Public Domain Software* entsteht, wenn ihr Urheber vollständig auf seine Rechte verzichtet und die Software der Allgemeinheit zur beliebigen Nutzung überläßt.[9] Ein Urheber offener Software nutzt hingegen seine Rechte ganz gezielt, um die weitere freie Verwendung der von ihm geschaffenen Software in der gewünschten Weise sicherzustellen. Insofern ist Public Domain Software weder offen noch proprietär. Ein Beispiel für eine Public Domain Software ist BIND.[10]

2. **Freeware** Als Freeware wird proprietäre Software bezeichnet, die dem Nutzer kostenfrei überlassen wird. Regelmäßig erhält der Nutzer zudem das

[8] Prominente Beispiele sind die *Apache Software License*, die *Artistic License* und die *MIT License*.

[9] Streng genommen ist der vollständige Verzicht auf sein Urheberrecht anders als im amerikanischen Rechtsraum in Deutschland nicht möglich. Die daraus erwachsenden Unterschiede sind aber hier nicht relevant (vgl. Jaeger und Metzger, 2002).

[10] BIND zählt zu den zentralen Bausteinen des Internet. Es besorgt die Zuordnung von Internetadressen (z.B. www.uni-mannheim.de) zu physisch vorhandenen Servern.

Recht, die Software in unveränderter Form weiterzugeben. Einsicht in den Quelltext wird dabei nicht gewährt. Zahlreiche Softwareunternehmen nutzen Freeware strategisch, sei es als Marketinginstrument oder im Rahmen einer extremen Penetrationsstrategie in Märkten mit Netzwerkeffekten. Ein Beispiel für eine Freeware ist der "Acrobat Reader" der Firma Adobe.

3. **Shareware** Shareware bezeichnet ein besonderes Marketing- und Vertriebskonzept für proprietäre Software. Dabei darf ein Nutzer die Software für eine gewisse Probezeit kostenfrei nutzen. Will er aber die Software nach Ablauf der Frist weiterhin verwenden, muß er Lizenzgebühren entrichten. Anderenfalls wird die weitere Nutzung oftmals technisch behindert oder der Funktionsumfang der Software reduziert. Des weiteren darf Shareware üblicherweise frei weiterverbreitet werden. Die wohl erfolgreichste Shareware aller Zeiten ist das Computerspiel Doom des Unternehmens "id software".

4. **Shared Source Software** Das Konzept der *Shared Source Software* wurde im Jahr 2001 vom Softwareunternehmen Microsoft als direkte Antwort auf offene Software vorgestellt. Ziel ist es, Microsoft die Vorteile offener Software zu erschließen, ohne einen Verlust an geistigem Eigentum erleiden zu müssen. Im Fall von Shared Source Software erhält der Nutzer das Recht, den Quelltext der Software einzusehen und zu verändern. Allerdings müssen diese Änderungen und alle Rechte daran an Microsoft zurückgegeben werden. Des weiteren ist die Weiterverbreitung derartiger Software untersagt.[11] Die erste Shared Source Software ist das Betriebssystem "Windows CE" von Microsoft.

Alle bislang vorgestellten Lizenzierungsarten für Software sind in Abbildung 2.1 kategorisiert zusammengefaßt. Als regulär wird dabei proprietäre Software bezeichnet, welche nicht unter eine der obigen Lizenzierungsarten 2 bis 4 fällt.

[11] Umfassende Informationen zu Shared Source Software bietet Microsoft unter der Adresse `www.microsoft.com/resources/sharedsource/default.mspx` an.

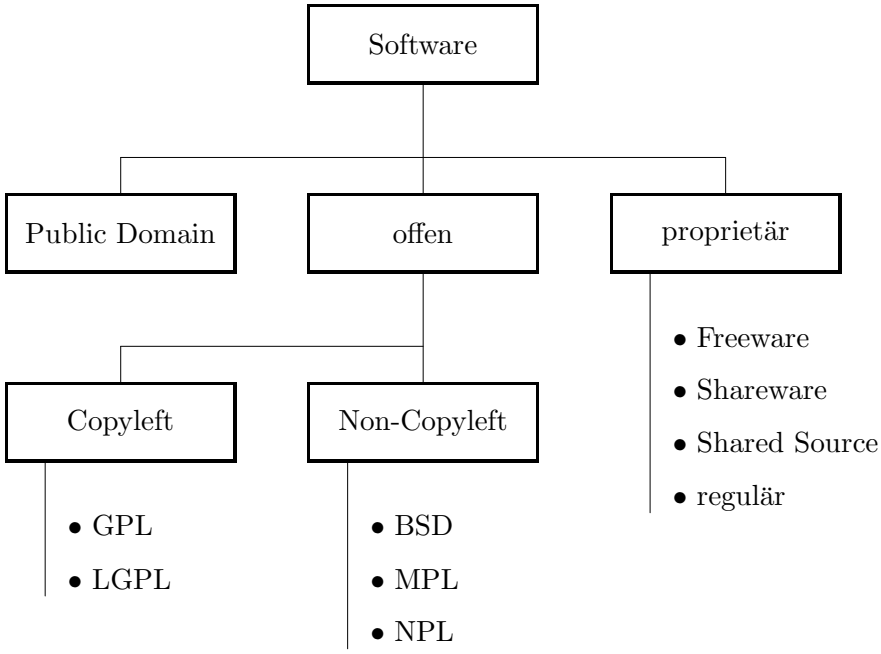


Abbildung 2.1: Lizenzierungsarten für Software

2.2 Soziokulturelle Aspekte offener Software

2.2.1 Entstehungsgeschichte

Nachfolgend wird in Anlehnung an Moody (2001), Raymond (2001), Feller und Fitzgerald (2002), Jaeger und Metzger (2002) sowie Fink (2003) die Entstehungsgeschichte offener Software geschildert. Es wird sich zeigen, daß offene Software ursprünglich eine unmittelbare Reaktion auf proprietäre Software war, die sich mit der Verbreitung des Internet zu einer neuen Form gemeinschaftlicher Softwareerstellung weiterentwickelt hat.

Bis in die späten 60er Jahre hinein wurde Software nicht als eigenständiges Gut, sondern als reine Ergänzung zu der damals relativ teuren Hardware angesehen. Standardisierte Massensoftware, wie heute allgemein gebräuchlich, wurde nicht verwendet. Stattdessen war es die Aufgabe angestellter Programmierer, auf den jeweiligen Einsatzbereich und die verfügbare Hardware individuell angepaßte Software zu erstellen. In der Folge entwickelte sich – von den Hardwareherstel-

lern gefördert – ein reger Austausch von Programmen und Quelltextbausteinen, der schließlich unter Programmierern eine allgemeine Kultur der Offenheit entstehen ließ, die sogenannte “Hacker-Kultur”. Im Jahr 1969 aber wurde gegen den damals dominanten Hardwarehersteller IBM ein Kartellverfahren angestrengt, in dessen Verlauf IBM die Bündelung von Hard- und Software aufgab.[12] Der Weg für eine eigenständige Softwareindustrie war bereitet.

Im Jahre 1969 begann in den “Bell Labs”, einem vom Telekommunikationsunternehmen AT&T betriebenen Forschungsinstitut, die Entwicklung des Betriebssystems Unix. Dieses Betriebssystem kann als das erste angesehen werden, welches konsequent als leicht auf neue Systeme übertragbare Zwischenschicht zwischen Hardware und einsatzspezifischer Anwendungssoftware konzipiert wurde. Obgleich es somit einen technologischen Meilenstein darstellt, wurde es zunächst nicht kommerziell verwertet, da es AT&T aufgrund ordnungspolitischer Erwägungen untersagt war, in anderen Märkten als dem für Telefonie tätig zu werden. Folglich wurde Unix von der oben beschriebenen Hacker-Kultur schnell aufgegriffen und, insbesondere im akademischen Umfeld, in vielfältiger Weise weiterentwickelt. Eine der besten und verbreitetsten Zusammenstellungen von Software rund um Unix war die seit dem Jahr 1977 an der Universität Berkeley betreute sogenannte Berkeley Software Distribution, auf die auch die im vorangegangenen Abschnitt vorgestellte BSD-Lizenz zurückgeht. BSD-Unix war in vielfacher Hinsicht bereits das, was man heute als ein offenes Softwareprojekt bezeichnet.

Der beinahe offene Charakter von Unix änderte sich im Jahr 1984 mit der Zerschlagung von AT&T. Der damit einhergehende Wegfall wettbewerbsrechtlicher Beschränkungen erlaubte AT&T die uneingeschränkte kommerzielle Verwertung von Unix. Da auch BSD-Unix an vielen Stellen Quelltext in der Urheberschaft von AT&T enthielt, mußte nun jeder, der BSD-Unix nutzen wollte, Lizenzgebühren entrichten. Ein erst in den 90er Jahren beigelegter Rechtsstreit mit AT&T setzte das BSD-Projekt weiter unter Druck. In Reaktion auf diese

[12] Das Verfahren wurde 1982 aufgrund einer völlig veränderten Situation eingestellt.

Entwicklung und verschiedene weitere Reibungspunkte zwischen der Hacker-Kultur und der erstarkenden Softwareindustrie initiierte Richard M. Stallman, damals Wissenschaftler am MIT, das GNU-Projekt, dessen Ziel die Rekonstruktion eines freien Unix-artigen Betriebssystems ohne Quelltext von AT&T war. GNU stellt dabei, gängigem Hackerhumor entsprechend, ein rekursives Akronym dar; es steht für "GNU's not Unix" und greift somit die Probleme von BSD-Unix unmittelbar auf. Im Jahr 1985 gründete Stallman des weiteren die gemeinnützige Free Software Foundation, um dem GNU-Projekt eine Heimat zu geben und seine Idee von freier Software allgemein zu fördern. Die FSF wiederum entwickelte im Jahr 1989 die GPL, um das GNU-Projekt langfristig vor der Vereinnahmung durch Hersteller proprietärer Software zu schützen. Das erste wahre offene Softwareprojekt war geboren.

Die FSF um Stallman machte bei der Entwicklung des GNU-Betriebssystems innerhalb der nächsten Jahre grundsätzlich große Fortschritte und stellte nahezu alle Teile eines Unix-artigen Betriebssystems bereit. Eines aber mißlang: die Entwicklung eines sogenannten Kernels, des Herzens eines jeden Betriebssystems.[13] Diese Lücke füllte, eher zufällig als geplant, der finnische Student Linus Torvalds, der 1991 zu Studienzwecken mit der Entwicklung eines rudimentären eigenen Kernels namens Linux begann.[14] Unter technologischen Gesichtspunkten war Linux sicherlich nichts besonderes; als revolutionär sollte sich aber erweisen, daß Torvalds konsequent das Internet nutzte, um andere in die Weiterentwicklung seines Kernels einzubinden. Er stellte den gesamten Quelltext im Netz zur Verfügung und lud in der Newsgroup `comp.os.minix`, einem Diskussionsforum für an der Entwicklung von Betriebssystemen Interessierte, zur Mitarbeit an Linux ein.[15] Die Resonanz auf diesen Aufruf war überwältigend.

[13] Inzwischen liegt der von der FSF entwickelte Kernel namens HURD in einer lauffähigen Version vor; nähere Informationen finden sich unter `www.gnu.org/software/hurd`.

[14] Der Name "Linux" wurde offensichtlich aus der spielerischen Verquickung von Torvalds Vornamen "Linus" und "Unix" gebildet.

[15] Der Originaltext dieses Aufrufs ist im Nachrichtenarchiv "Google Groups" unter `groups-beta.google.com/group/comp.os.minix/msg/b813d52cbc5a044b` nachzulesen.

Eine stetig wachsende Zahl an Programmierern begann, in vielfältiger Weise zu Linux beizutragen, und bereits im Jahr 1994 konnte die erste stabile und für den allgemeinen Einsatz taugliche Version von Linux veröffentlicht werden. In den nächsten Jahren wurden zahlreiche weitere offene Softwareprojekte, beispielsweise Apache, KDE und GNOME,[16] gegründet und nach dem Vorbild von Linux in einem offenen, internetgestützten Prozeß vorangetrieben. Im Jahr 1997 schließlich veröffentlichte Eric S. Raymond, selbst seit den 80er Jahren in verschiedenen offenen Softwareprojekten aktiv, seinen einflußreichen Essay "The Cathedral and the Bazaar".[17] In diesem analysiert er den im Rahmen des Linux-Projekts erstmals erprobten offenen Entwicklungsprozeß und arbeitet seine Vorteile gegenüber den stärker strukturierten, üblicherweise in Softwareunternehmen praktizierten Methoden heraus. Bildreich prägt Raymond für diese beiden gegensätzlichen Entwicklungsphilosophien die Bezeichungen Basar- bzw. Kathedralenmethode. Es ist auf Raymonds Essay zurückzuführen, daß heute unter offener Software nicht nur eine Lizenzierungsart, sondern auch ein Paradigma der Softwareentwicklung, eben die Basarmethode, verstanden wird.

Im Jahr 1998 gelang offener Software der Durchbruch in den Mainstream. Angeregt durch "The Cathedral and the Bazaar" beschloß das Unternehmen Netscape, sein bis dato proprietäres Produkt "Navigator" als offene Software zu veröffentlichen. Ziel war es, die Vorzüge der Basar-Methode zu nutzen, um den für die Unternehmensstrategie wichtigen Navigator im gegen Microsoft geführten sogenannten Browser-Krieg wieder auf die Gewinnerstraße zu führen.[18] Um den spezifischen Problemen gerechtzuwerden, die sich aus der Überführung einer proprietären in eine offene Software ergeben, entwickelte Netscape überdies gemeinsam mit Schlüsselfiguren der Hacker-Kultur die MPL und NPL. Von diesem Erfolg angespornt suchten die seitens der Hacker-Kultur

[16] Weitere Angaben zu diesen Projekten finden sich unter `www.apache.org`, `www.kde.org` bzw. `www.gnome.org`.

[17] Eine aktualisierte Version dieses Essays ist in Raymond (2001) abgedruckt.

[18] Der Browser-Krieg ging für Netscape schnell verloren. Unter dem Dach des offenen Mozilla-Projekts hat ihn der Navigator aber überlebt.

Jahr	Ereignis
1969	Beginn der Entwicklung von Unix bei AT&T
1977	Gründung der Berkeley Software Distribution
1984	Zerschlagung von AT&T
1985	Gründung der Free Software Foundation (FSF)
1989	Einführung der GPL
1991	Beginn der Entwicklung von Linux
1994	Veröffentlichung der Version 1.0 von Linux
1997	Veröffentlichung von "The Cathedral and the Bazaar"
1998	Netscape veröffentlicht den Navigator unter einer offenen Lizenz
	Prägung des Begriffs "Open Source Software"
	Gründung der Open Source Initiative (OSI)
2001	IBM investiert 1 Mrd. $ in Linux-Aktivitäten

Tabelle 2.3: Wichtige Ereignisse in der Entstehungsgeschichte offener Software

an der Offenlegung des Navigator Beteiligten nach Wegen, die Verwendung offener Software und der Basar-Methode auch in anderen Bereichen der Softwareindustrie zu propagieren. Da sie dem bis dahin gebräuchlichen Begriff "Free Software" wirtschaftsfeindliche Konnotationen zuschrieben, prägten sie die neutralere Bezeichnung "Open Source Software" und gründeten als Lobby die gemeinnützige Open Source Initiative. Die Bemühungen der OSI, offene Software in der Industrie zu etablieren, bekamen bald unerwarteten Rückenwind. In internen Memos des Softwareunternehmens Microsoft, nach dem Tag ihres ungewollten Publikwerdens als Halloween-Dokumente bezeichnet, wird offene Software als große Bedrohung für das Unternehmen identifiziert und die bemerkenswerte Qualität der mit der Basar-Methode entwickelten Software gelobt.[19] Die Folge dieses Ritterschlags war ein rasanter Anstieg des öffentlichen Interesses. Im weiteren nahm auch die Akzeptanz offener Software in der Industrie stetig zu. Im Januar 2001 schließlich kündigte IBM an, allein in jenem Jahr über 1 Mrd. $

[19]Die Halloween-Dokumente sind unter `www.opensource.org/halloween/` einzusehen.

in Linux-bezogene Aktivitäten zu investieren und die Unternehmensstrategie entsprechend anzupassen (vgl. Wilcox, 2000). Offene Software war erwachsen geworden.

2.2.2 Exemplarische offene Softwareprojekte

Wie gesehen haben Betriebssysteme bei der Entstehung offener Software eine zentrale Rolle gespielt. Dennoch ist offene Software beileibe nicht nur in diesem Bereich anzutreffen. Mertens et al. (2005) unterscheiden nach ihrem Verwendungszweck vier Kategorien von Software,

- *System- und Netzwerkmanagementsoftware*, welche in erster Linie die technische Infrastruktur für die Ausführung weiterer Software bereitstellt,
- *Informationsmanagementsoftware*, welche die Speicherung und Verwaltung großer Datenbestände besorgt,
- *Anwendungssoftware*, welche Endnutzern unmittelbar bei der Lösung spezifischer Aufgaben hilft,
- sowie *Programmentwicklungssoftware und Middleware*, welche der Entwicklung neuer Software und der Verknüpfung mehrerer Programme zu größeren, anwendungsspezifischen Systemen dient.

Nachfolgend wird die Vielfalt offener Software anhand von vier Entwicklungsprojekten illustriert, die jeweils einer der angegebenen Kategorien entstammen.[20] Die Projekte sind so ausgewählt, daß möglichst viele weitere Facetten offener Software deutlich werden; Schwerpunkte der Darstellung bilden dabei die Entstehungsgeschichte, die Organisationsstruktur sowie der Markterfolg[21] der Software.

[20] Brügge et al. (2004, S. 27) ordnen zahlreiche weitere offene Softwareprojekte in diese Kategorien ein.

[21] Grundsätzlich beziehen sich in dieser Arbeit alle angegebenen Marktanteile auf die Anzahl an Installationen, da die monetäre Bewertung der Absatzzahlen offener Software durch das Recht auf Weiterverbreitung ohne Lizenzgebühren nicht sinnvoll ist.

2.2.2.1 Linux-Kernel

Die Entstehung und die herausragende Bedeutung des in die Kategorie "System- und Netzwerkmanagementsoftware" fallenden Linux-Kernels wurden bereits im vorigen Abschnitt dargestellt. Nachfolgend wird in Anlehnung an Moon und Sproull (2002) sowie Lee und Cole (2003) die informelle, prototypische Organisationsstruktur des dazugehörigen Entwicklungsprojekts beschrieben.

Das zentrale Kommunikationsinstrument des Projekts ist die sogenannte *Linux Kernel Mailing List* (LKML), die im Prinzip wie eine große, strukturierte Pinnwand arbeitet und die grundsätzlich jeder einsehen und nutzen darf. Allerdings hat das Projekt für die Nutzung der LKML einige Regeln aufgestellt, deren Mißachtung schlicht zu Nichtbeachtung führt. Des weiteren steht in den sogenannten *Linux Kernel Archives* (LKA) permanent der Quelltext der aktuellen Version des Linux-Kernels bereit.[22] Ausgehend vom in den LKA vorgehaltenen Quelltext kann jedermann auf der LKML Fehler (sogenannte Bugs) melden und Fehlerkorrekturen bzw. Verbesserungen des Quelltextes (sogenannte Patches) vorschlagen. Diese werden wiederum auf der LKML in einem transparenten Prozeß ausgiebig begutachtet und weiter verbessert. Bei Gefallen wählt dann ein mit Schreibrechten auf die LKA ausgestatteter sogenannter *Maintainer* auf der Grundlage des Begutachtungsprozesses einen Patch zur Aufnahme in die nächste Version von Linux auf. Alle Maintainer werden dabei aufgrund der besonderen Qualität ihrer Beiträge in der Vergangenheit ernannt und sind für einzelne Komponenten des Kernels zuständig. Die finale Entscheidung über die Aufnahme eines Patches in den Kernel behält sich Torvalds als Projektleiter[23] zwar vor, im Regelfall folgt er aber der Empfehlung der Maintainer. Schließlich werden die LKA aktualisiert und der beschriebene Prozeß beginnt erneut. Des weiteren legen Torvalds und die Maintainer die allgemeine Stoßrichtung des Projekts fest und implementieren grundlegend neue Funktionen.

[22] Die Regeln für die Nutzung der LKML sind unter `www.kernel.org/pub/linux/docs/lkml/`, die LKML selbst unter `vger.kernel.org` und die LKA unter `www.kernel.org` zu finden.

[23] Torvalds hält überdies die Rechte am eingetragenen Markenzeichen "Linux".

Lee und Cole (2003) zeigen auf der Grundlage der LKA, daß in den Jahren 1995 bis 2000 durchschnittlich alle 5 Tage eine neue Entwicklungsversion von Linux veröffentlicht wurde. Des weiteren schätzen sie anhand der LKML die Anzahl derer, die in den unterschiedlichen Funktionen in demselben Zeitraum zur Entwicklung des Kernels beigetragen haben. Neben Torvalds als einzigem Projektleiter gab es 121 Maintainer, 2.605 Autoren von Patches sowie 1.562 Projektteilnehmer, die lediglich Bugs gemeldet haben.

Linux ist zweifelsfrei auch wirtschaftlich die bedeutendste offene Software.[24] Bozman et al. (2004) geben den weltweiten Umsatz von Produkten, die gemeinsam mit Linux eingesetzt werden, für das Jahr 2004 mit 15 Mrd. $ an und schätzen, daß sich dieser Umsatz auf 35 Mrd. $ im Jahr 2008 erhöhen wird. Diese Zahlen spiegeln sich auch in den hohen Marktanteilen von Linux in einigen Betriebssystemmärkten wider. So beziffert Gonsalves (2003) den Anteil von Linux bei vorinstallierten Betriebssystemen auf neuen *Servern* im Jahr 2002 auf 23 %, wohingegen Windows 55 %, Unix 11 % und NetWare 10 % zukommen; Bozman et al. (2004) sehen den Marktanteil von Linux auf Servern bei 20 % im Jahr 2004 und prognostizieren ein Wachstum dieses Anteils auf 28 % im Jahr 2008. Diese Angaben sind allerdings mit gewissen Meßungenauigkeiten behaftet. Sehr exakt sind hingegen die folgenden Zahlen zur Verbreitung verschiedener Betriebssysteme auf *Webservern*, da derartige Server zwangsläufig öffentlich zugänglich sind und mit technischen Mitteln genau analysiert werden können. Demnach war im April 1999 Linux auf 31 %, Windows auf 24 %, Solaris auf 17 % und BSD-Unix auf 15 % der öffentlichen Webserver installiert.[25] Laut King (2002) soll sich der Anteil von Linux in diesem Teilmarkt auf 41 % im Jahr 2005 erhöhen. Kaum Verbreitung gefunden hat Linux hingegen im Markt für *Desktop-PCs*. Hier dominierte im Jahr 2002 nach Gonsalves (2003) Windows mit 94 % Marktanteil vor Linux und "Mac OS" mit jeweils 3 %. Bozman et al.

[24] Der Linux-Kernel allein ist natürlich kein marktfähiges Produkt. Insofern beziehen sich alle folgenden Angaben auf das um den Kernel gebaute Betriebssystem.

[25] Die genaue technische Vorgehensweise, nach welcher diese Zahlen ermittelt wurden, ist unter `www.leb.net/hzo/ioscount` angegeben.

(2004) erwarten aber immerhin eine Verdopplung des Anteils von Linux auf 7 % im Jahr 2008. Insgesamt zeigt sich, daß Linux in Einsatzfeldern abseits der Endnutzer ein ernstzunehmender Wettbewerber des proprietären Windows ist, während es im Endnutzerbereich bisher kaum Verbreitung gefunden hat.

2.2.2.2 Apache

Die nach Linux profilierteste offene Software ist wohl der in die Kategorie "Informationsmanagementsoftware" einzuordnende Webserver Apache. Webserver dienen der Bereitstellung von mittels eines Webbrowsers angeforderter Internetseiten und haben somit eine zentrale Bedeutung für den Betrieb des World Wide Web.

Apache geht auf einen frühen, vom Forschungsinstitut NCSA entwickelten Webserver zurück, dessen Quelltext frei verfügbar war.[26] Als das NCSA im Jahr 1994 die Entwicklung seines Webservers aufgab, übernahmen einige Nutzer aus reiner Notwendigkeit heraus die Pflege und Weiterentwicklung der Software. Um diese Kerngruppe früher Entwickler bildete sich bald eine Gemeinschaft, deren Organisationsstruktur der des Linux-Projekts weitestgehend entsprach. Im Jahr 1999 aber wurde diese Struktur anders als im Fall des Linux-Kernels in der gemeinnützigen *Apache Software Foundation* (ASF) formalisiert. Alle Projektteilnehmer gehören einer von fünf Hierarchieebenen an, vom einfachen Nutzer, der lediglich Bugs melden und Verbesserungsvorschläge unterbreiten darf, bis hin zum ASF-Mitglied mit weitreichenden Entscheidungsbefugnissen. Der Aufstieg innerhalb dieser Hierarchie erfolgt dabei gemäß der technischen Fähigkeiten nach recht klar spezifizierten Regeln. ASF-Mitglied schließlich wird eine natürliche Person allein auf Vorschlag durch ein bestehendes Mitglied mit anschließender Wahl. Die Gemeinschaft aller ASF-Mitglieder ist das oberste Entscheidungsorgan des Projekts.[27] Hier spiegelt sich die Tatsache wider, daß Apache anders als Linux mehrere Gründer hat und somit eine Formalisierung

[26] Unter `www.apache.org/foundation/how-it-works.html` sind genaue Informationen zu Apache verfügbar.

[27] Im Januar 2005 gab es laut `www.apache.org/foundation/members.html` 112 aktive ASF-

der Entscheidungsprozesse erforderlich war. Des weiteren stellt die ASF eine umfangreiche Infrastruktur für das Projekt bereit, nimmt als eigene Rechtspersönlichkeit Spenden entgegen, schützt die einzelnen Entwickler vor Rechtsstreitigkeiten und hütet den Markennamen "Apache".

Der Apache-Webserver ist seit Jahren unangefochtener Marktführer. Im Januar 2005 wurde er auf 68 % aller öffentlich zugänglichen Server eingesetzt; Platz 2 belegte mit 21 % der proprietäre Webserver IIS von Microsoft.[28]

2.2.2.3 OpenOffice.org

Die Anwendungssoftware OpenOffice.org ist als vollwertige Kollektion von Büroprogrammen mit dem weitverbreiteten "Microsoft Office" vergleichbar. Die Wurzeln der Software reichen bis zum von der Firma StarDivision seit Mitte der 80er Jahre entwickelten proprietären StarOffice zurück.[29] Im Jahr 1999 wurde StarDivision zur strategischen Abrundung des Produktportfolios vom Unternehmen Sun aufgekauft. Ein Jahr später veröffentlichte Sun große Teile von StarOffice unter dem Namen "OpenOffice.org" als offene Software. Dabei erlaubt es die Lizenzierung Sun, OpenOffice.org um proprietäre Elemente zu erweitern und unter dem alten Namen StarOffice zu vertreiben. Des weiteren stellte Sun eine technische Infrastrukur und eine formale, basisdemokratisch konzipierte Organisationsstruktur bereit, um die schnelle Entstehung einer Entwicklergemeinschaft rund um den veröffentlichten Quelltext zu fördern.

OpenOffice.org ist in zahlreiche Teilprojekte gegliedert, in denen jeweils die eigentliche Entwicklung einer exakt spezifizierten Komponente des Gesamtsy-

Mitglieder. Davon waren 70 bei Informationstechnologieunternehmen angestellt, 24 freiberuflich und 3 an Universitäten beschäftigt. Die Affiliation der verbleibenden 15 Mitglieder konnte nicht festgestellt werden.

[28] Diese Zahlen werden jeden Monat vollautomatisch vom Marktforschungsunternehmen Netcraft ermittelt. Aktuelle Ergebnisse sind unter `www.netcraft.com/Survey/Reports/` zu finden.

[29] Die Entstehungsgeschichte von OpenOffice.org ist unter `www.openoffice.org/about.html` nachzulesen.

stems vorgenommen wird. Innerhalb eines jeden Teilprojekts existiert wie beim Apache-Projekt eine formelle Hierarchie mit einem Projektleiter an der Spitze.[30] Der Projektleiter wird dabei von allen Teilprojektmitgliedern gewählt und darf verdienten Entwicklern Schreibrechte auf den Quelltext des Teilprojekts gewähren. Das oberste Entscheidungsgremium des Projekts ist aber das sogenannte "Community Council".[31] Dieses Gremium legt die langfristige Strategie von OpenOffice.org fest, vertritt das Projekt nach außen und fungiert als Schiedsrichter bei Kompetenzstreitigkeiten innerhalb der Teilprojekte. Es umfaßt neun, basisdemokratisch gewählte Miglieder aus verschiedenen Bereichen des Gesamtprojekts, von denen mindestens zwei Angestellte von Sun sind.[32] OpenOffice.org gilt als gelungene Symbiose zwischen einem Unternehmen und einer offenen Entwicklergemeinschaft klassischer Prägung.

Laut Thibodeau (2004) sah das Marktforschungsunternehmen Gartner Group den gemeinsamen Marktanteil von OpenOffice.org und StarOffice im Jahr 2004 bei unter 5 %; klarer Marktführer ist seit Jahren mit Marktanteilen weit über 90 % "Microsoft Office".

2.2.2.4 Eclipse

Eine bekannte offene Software aus der Kategorie "Programmentwicklungssoftware und Middleware" ist Eclipse. Diese Software wurde im Jahr 2001 als bereits voll ausgereiftes Produkt von IBM unter eine offene Lizenz gestellt.[33] Des weiteren gründete IBM mit einigen weitere Unternehmen ein Konsortium, welches die Weiterentwicklung von Eclipse vorantreiben sollte. Drei Jahre später wurde

[30] Genaue Angaben zu den Organisations- und Entscheidungsstrukturen der Teilprojekte sind unter `www.openoffice.org/dev_docs/guidelines.html` zu finden.

[31] Die Satzung dieses Gremiums steht unter `council.openoffice.org/CouncilProposal.html`.

[32] Sun rechtfertigt seinen besonderen Einfluß, indem es nach wie vor signifikante Beiträge zur Weiterentwicklung von OpenOffice.org leistet. So waren im Januar 2005 laut `projects.openoffice.org/accepted.html` 23 von insgesamt 43 Entwicklern mit Leitungsfunktion in technischen Teilprojekten bei Sun beschäftigt.

[33] Die Geschichte von Eclipse wird unter `www.eclipse.org/org/main.html` beschrieben.

dieses Konsortium in die gemeinnützige *Eclipse Foundation* umgewandelt. Dennoch ist das Eclipse-Projekt im Kern nach wie vor ein reiner Zusammenschluß von Unternehmen, wobei formal zwischen strategischen Entwicklern, strategischen Nutzern und Anbietern komplementärer Produkte unterschieden wird.

In der Folge wird das Eclipse-Projekt eher wie ein klassisches Konsortium denn wie ein offenes Softwareprojekt geführt.[34] Zwar ähnelt es innerhalb seiner Teilprojekte den oben beschriebenen offenen Entwicklungsprojekten sehr stark, verfügt also über eine frei zugängliche Entwicklerhierarchie und transparente Prozesse. Die Teilprojektleitung wird aber letztlich immer vom obersten Entscheidungsgremium, dem sogenannten "Board of Directors", eingesetzt. Das Board wiederum besteht in seiner großen Mehrheit aus Vertretern der Mitgliedsunternehmen, wobei die strategischen Entwickler und Nutzer ein besonderes Gewicht haben.[35] Das Eclipse-Projekt wird somit anders als pure offene Softwareprojekte nicht aus sich selbst heraus geführt.

Auf der Basis einer Umfrage unter Entwicklern schätzte der Softwarehersteller QA Systems (2003) den Anteil von Eclipse am Markt für Java-Entwicklungsumgebungen im Jahr 2003 auf 45 % vor dem proprietären JBuilder mit 16 %.

2.2.3 Werte und Normen

Offensichtlich handelt es sich bei offener Software um ein gewachsenes Phänomen, dessen Wesen nicht allein technische und lizenzrechtliche, sondern ganz wesentlich auch soziokulturelle Aspekte ausmachen. Nachfolgend werden die wichtigsten dieser Aspekte, nämlich die Werte und Normen, welche die reibungslose Zusammenarbeit in offenen Softwareprojekten ermöglichen, kurz vorgestellt.

---

[34] Unter `www.eclipse.org/org/documents/main.html` ist die Organisation des Eclipse-Projekts sehr genau beschrieben.

[35] Im Januar 2005 waren laut `www.eclipse.org/org/directors.html` 10 von 13 Board-Mitgliedern in der Eigenschaft als Repräsentanten eines strategischen Mitgliedsunternehmens vertreten. Überdies waren die drei verbleibenden Board-Mitglieder alle bei einem Mitgliedsunternehmen angestellt.

Die eigentliche Entwicklungsarbeit wird im wesentlichen von drei ineinandergreifenden Prinzipien bestimmt, denen des Meritokratismus, der Transparenz und des offenen Zugangs:[36] So ist der Einfluß von Entwicklern auf den offiziellen Quelltext stets eng an deren bisherige Leistungen gekoppelt. Des weiteren werden sämtliche Beiträge zum Projekt, von einer kleinen Anregung bis hin zu einer umfangreichen Quelltexterweiterung, namentlich veröffentlicht und in der Regel auch dauerhaft archiviert. Und schließlich steht es jedermann frei, an einem Projekt mitzuarbeiten und höhere Ränge innerhalb der Entwicklerhierarchie zu erklimmen.

Ein notwendige Bedingung für das Funktionieren einer Meritokratie ist die Möglichkeit des Einzelnen, Reputation aufzubauen und zu erhalten. Neben dem Prinzip der Transparenz hat die Hacker-Kultur zu diesem Zweck laut Raymond (2001) zahlreiche weitere Normen entwickelt; beispielsweise gilt es kulturell als Kapitalverbrechen, den Namen eines ehemaligen Projektmitglieds aus einem Projekt zu tilgen. Wiederholte Verstöße gegen diese Normen führen regelmäßig zu sozialer Ächtung oder gar zum Ausschluß aus einem Projekt.[37]

Bezroukov (1999) und Kelty (2001) vergleichen offene Softwareentwicklung mit dem wissenschaftlichen Arbeitsprozeß und heben hervor, wie sehr sich beide Vorgehensweisen in ihrer Betonung von Offenheit und transparenter Begutachtung durch Peers ähneln. In einer ähnlichen Betrachtung identifiziert Edwards (2001) offene Entwicklungsprojekte als eine spezielle Ausprägung sogenannter epistemischer, also auf Erkenntniszuwachs zielender Gemeinschaften. Demnach teilen Mitglieder an offenen Entwicklungsprojekten regelmäßig normative Grundüberzeugungen, Methodenkompetenz und Bewertungs- und Vergleichskriterien für konkurrierende Lösungen. Die Hacker-Szene kann ihre uni-

[36] Ausdrücklich als tragend werden diese Prinzipien etwa unter `www.eclipse.org/org/documents/Eclipse%20Development%20Process%202003_11_09%20FINAL.pdf` im Rahmen des Eclipse-Projekts anerkannt.

[37] Auch Verstöße gegen gewisse Umgangsformen, die den für die Kommunikation über das Internet gebräuchlichen sogenannten "Netiquette" sehr ähnlich sind, werden nach Bergquist und Ljungberg (2001) schnell bestraft.

versitären Wurzeln also nicht verleugnen.

Laut Brügge et al. (2004) schützt der von den gemeinsamen Werten und Normen induzierte starke Gemeinschaftssinn offene Softwareprojekte auch vor sogenanntem Forking, das heißt, der Spaltung eines in zwei konkurrierende Projekte. Sprechen nicht wichtige Gründe dafür, gilt Forking in der Hacker-Kultur üblicherweise als unerwünscht, da es schnell zu einer Zersplitterung der Entwicklungsanstrengungen und zu inkompatiblen Produkten führt. Offene Lizenzen selbst behindern Forking offensichtlich nicht; rein rechtlich darf selbst ein Außenstehender ohne Meriten innerhalb eines Projekts dessen Quelltext beliebig modifizieren und als neue Variante der Software verbreiten. Faktisch wird es ihm aber kaum möglich sein, hinlängliche Unterstützung für die unerläßliche Weiterentwicklung seiner Variante zu gewinnen. Raymond (2001) bezeichnet offene Softwareentwicklung daher als theoretisch promiskuitiv, aber praktisch puritanisch.[38]

Im Hinblick auf die eigentliche Zusammenarbeit verfügt die Hacker-Kultur also über eine Vielzahl gemeinsamer Werte und Normen. Eine Spaltung in zwei Lager hat sich aber aus der Frage ergeben, warum Software offen sein sollte.[39] Auf der einen Seite stehen die Pragmatiker, welche die technischen Vorteile der Basar-Methode sowie offener Standards betonen. Sie haben den Begriff "Open Source Software" geprägt und befürworten Kooperationen mit Unternehmen. Auf der anderen Seite finden sich die Fundamentalisten, welche vornehmlich freiheitliche Aspekte in den Vordergrund stellen. Sie ziehen die Bezeichnung "Free Software" vor und stehen Kooperationen mit Unternehmen sehr kritisch gegenüber. Lakhani und Wolf (2005) rechnen auf der Grundlage einer umfangreichen Umfrage rund 20 % aller Entwickler offener Software dem fundamentalistischen Lager zu; Gosh et al. (2002) ermitteln hingegen einen Anteil von fast 50 %.

[38] Dennoch existieren einige prominente Beispiele für Forking, etwa der Zerfall von BSD-Unix in FreeBSD, NetBSD und OpenBSD sowie die beiden Texteditoren "GNU Emacs" und XEmacs.

[39] Raymond (1999) und Stallman (1999) stellen die Positionen beider Lager ausführlich dar.

2.3 Teilhaber an offenen Softwareprojekten

2.3.1 Entwickler

Die offensichtlichste Gruppe von Teilhabern an offenen Softwareprojekten bilden ihre Entwickler. Dabei umfaßt diese Gruppe allerdings nicht nur jene, welche im engen Sinn programmieren, sondern auch die, welche ein Entwicklungsprojekt auf andere Art durch ihren Arbeitseinsatz mittragen, beispielsweise Designer, Tester und Autoren von Inhalten und Dokumentationsmaterial.

2.3.1.1 Einzelpersonen

Es ist sehr aufschlußreich, die Entwickler offener Software etwas genauer kennenzulernen. Dazu werden in diesem Abschnitt einige vornehmlich soziodemographische Merkmale präsentiert, welche Rückschlüsse auf das Qualifikationsniveau und die geographische Herkunft der Entwickler sowie den Umfang der geleisteten Arbeit und den Hauptanstoß dafür zulassen. Die Grundlage der Darstellung bilden mehrere Umfragen unter Entwicklern offener Software, die in Tabelle 2.4 zusammengestellt sind. Die Umfragen lassen sich dabei in zwei Klassen einteilen: Die Arbeiten von Jørgensen (2001b), Hars und Ou (2002), Hertel et al. (2003) und Lakhani und Wolf (2005) konzentrieren sich auf wohlabgegrenzte Personenkreise. Dieses ermöglicht es den Autoren sicherzustellen, daß es sich bei den Befragten tatsächlich um Entwickler im oben angegebenen Sinn handelt, begrenzt aber die Anzahl der Befragten mitunter erheblich. Die Studien von Robles et al. (2001) und Gosh et al. (2002) können hingegen mit weit mehr befragten Personen aufwarten, leiden aber unter dem Problem der unscharfen Abgrenzung der Auskunftspersonen. Beide Studien haben ihre Daten über Fragebögen im Internet erhoben, die zwar an einschlägigen virtuellen Orten beworben wurden, grundsätzlich aber für jedermann zugänglich waren. Insofern können die Autoren nur hoffen, daß sich zumindest in der Mehrzahl echte Entwickler beteiligt haben.

Zunächst werden solche Merkmale diskutiert, welche Aussagen über das Quali-

Autor(en)	Beschreibung der Befragten	Anzahl
Jørgensen (2001b)	Entwickler von FreeBSD mit Schreibrechten auf den offiziellen Quelltext	72
Robles et al. (2001)	Leser der Internetseiten `www.heise.de`, `www.barrapunto.com` und `slashdot.org`	5.478
Gosh et al. (2002)	Teilnehmer an Diskussionsforen zu diversen offenen Softwareprojekten	2.784
Hars und Ou (2002)	Teilnehmer ausgewählter Diskussionsforen zur Entwicklung offener Software	79
Hertel et al. (2003)	Entwickler des Linux-Kernels und Leser der Linux Kernel Mailing List	141
Lakhani und Wolf (2005)	Entwickler 287 verschiedener, auf SourceForge.net angesiedelter Projekte	684

Tabelle 2.4: Umfragen unter Entwicklern offener Software

fikationsniveau offener Softwareentwickler erlauben. Einen ersten Anhaltspunkt bietet das Alter der Entwickler, welches in den Untersuchungen von Hertel et al. (2003), Hars und Ou (2002) sowie Lakhani und Wolf (2005) durchschnittlich 30 Jahre, in denen von Gosh et al. (2002) und Robles et al. (2001) hingegen 27 Jahre beträgt.[40] Des weiteren können im Falle dreier Studien auch Aussagen über die Verteilung des Alters gemacht werden (vgl. Tabelle 2.5). Es zeigt sich, daß die offene Softwarebewegung ganz wesentlich von Entwicklern zwischen 20 und 40 Jahren getragen wird.

Zudem zeigen Robles et al. (2001), Gosh et al. (2002) sowie Hars und Ou (2002), daß Entwickler offener Software überdurchschnittlich gut ausgebildet sind. So haben nach den genannten Studien zwischen 70 % und 80 % der Befragten mindestens einen Universitätsabschluß. Demgegenüber verfügen nach einer Un-

[40] Die Differenz von 3 Jahren ist wohl darauf zurückzuführen, daß die drei erstgenannten Untersuchungen nur solche Entwickler betrachten, die sich bereits einen gewissen Rang in der für sie relevanten Hierarchie erarbeitet haben, wohingegen die beiden letztgenannten auch untere Chargen einbeziehen.

Alter†	Robles et al. (2001)	Hars und Ou (2002)	Gosh et al. (2002)
10 - 19	10 %	4 %	10 %
20 - 29	62 %	50 %	60 %
30 - 39	23 %	35 %	22 %
40 - 49	4 %	10 %	6 %
≥ 50	1 %	1 %	2 %

†in Jahren

Tabelle 2.5: Umfragen unter Entwicklern offener Software

tersuchung der OECD (2004) nur 32 % der Bevölkerung ihrer Mitgliedsstaaten über einen universitären Grad.

Überdies zielt die gute Ausbildung der meisten offenen Softwareentwickler ganz deutlich auf eine berufliche Tätigkeit mit starkem Bezug zur Informationstechnologie. Dieses zeigt die Untersuchung von Gosh et al. (2002), welche die Haupttätigkeit aller Befragten ermittelt (vgl. Tabelle 2.6). So arbeiten über 80 % aller Entwickler entweder in einem IT-Unternehmen oder im einschlägigen universitären Umfeld, sei es als Wissenschaftler oder als Studenten; immerhin 44,5 % sind zudem als Softwareingenieure oder Programmierer ganz unmittelbar mit professioneller Softwareentwicklung befaßt. Weitere Studien stützen diese Ergebnisse im Rahmen ihres begrenzten Auflösungsvermögens. Die von Hertel et al. (2003) Befragten sind zu 72 % Angestellte, zu 23 % Studenten und zu 5 % Arbeitslose; die von Robles et al. (2001) sind zu rund 80 % im IT-Umfeld tätig. Diese IT-bezogene Arbeit schlägt sich auch in der Programmiererfahrung der Entwickler nieder: Sie beträgt laut Lakhani und Wolf (2005) im Durchschnitt 11,9 Jahre bei einer Standardabweichung von 7,0 Jahren.

Insgesamt zeigt sich klar, daß die Entwickler offener Software mehrheitlich hochqualifizierte Profis sind. Das Bild vom pickligen Teenager, welcher in Ermangelung einer Freundin vereinsamt vor dem Computer hockt und offene Software programmiert, hält einer eingehenden Betrachtung nicht stand.[41]

[41]Das Klischee entspricht nur insoweit der Realität, als die Entwickler offener Software quasi

Haupttätigkeit	mit IT-Bezug	ohne IT-Bezug	Gesamt
Ingenieur	33,3 %	3,2 %	36,5 %
Programmierer	11,2 %	-	11,2 %
Berater	9,8 %	0,6 %	10,4 %
Kaufmännischer Angestellter	3,5 %	0,3 %	3,8 %
Universitätswissenschaftler	5,0 %	4,3 %	9,3 %
Student	15,8 %	5,1 %	20,9 %
Sonstiges	5,2 %	2,7 %	7,9 %
Gesamt	83,8 %	16,2 %	100,0 %

Tabelle 2.6: Tätigkeiten von Entwicklern offener Software (Gosh et al., 2002)

Des weiteren ist die für die Entwicklung offener Software aufgewendete Zeit mitunter ganz erheblich. So reicht die durchschnittliche Arbeitszeit pro Woche von 11,7 Stunden in der Studie von Gosh et al. (2002) über 11,8 und 12,4 bei Robles et al. (2001) bzw. Hertel et al. (2003) bis zu 14,1 Stunden bei Lakhani und Wolf (2005). Allerdings variiert der Zeiteinsatz unter den Entwicklern sehr stark. Alle Untersuchungen zeigen übereinstimmend, daß über 50 % der Beteiligten deutlich unter 10, knapp 10 % hingegen deutlich über 40 Stunden pro Woche aufwenden. Dementsprechend konzentriert sich das Gros der insgesamt geleisteten Arbeit nur auf einen kleinen Anteil der beteiligten Entwickler. Dieses ist auch aus Abbildung 2.2 ersichtlich, welche für die Untersuchungen von Robles et al. (2001), Gosh et al. (2002) sowie Lakhani und Wolf (2005)[42] die Verteilung des Arbeitseinsatzes der Entwickler jeweils als Lorenz-Kurve zeigt.[43] Die für alle drei Untersuchungen quasi identische, stark durchhängende Form aller Kurven zeigt deutlich die hohe Konzentration; entsprechend betragen die

alle männlich sind. Ob sie keine Freundinnen haben, wurde hingegen bislang nicht erhoben.

[42] Für diese Untersuchung wurde zum Teil auch auf Lakhani et al. (2002) zurückgegriffen, da diese früheren Veröffentlichung einige Rohdaten enthält, die in Lakhani und Wolf (2005) nicht angegeben werden.

[43] Da die zugrundeliegenden Daten nur in klassierter Form vorlagen, wurden die Lorenzkurven unter der Annahme von Gleichverteilungen innerhalb der einzelnen Klassen ermittelt.

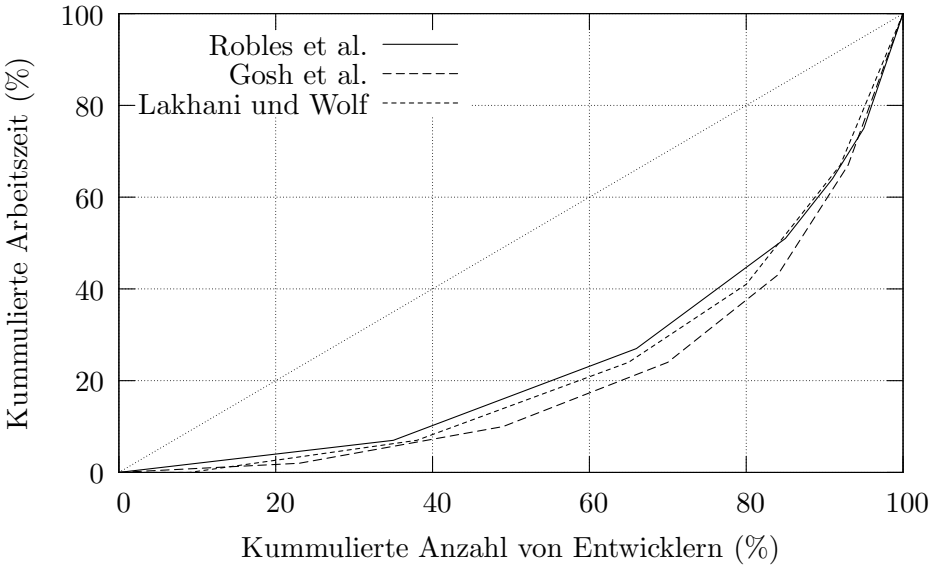


Abbildung 2.2: Lorenz-Kurven zum Zeiteinsatz der Entwickler

Gini-Koeffizienten[44] 0,50, 0,58 bzw. 0,54.

Insgesamt erscheint es sinnvoll, die Entwickler nach ihrem Arbeitseinsatz vereinfachend in zwei Gruppen einzuteilen, *periphäre Entwickler*, die zeitlich nur einen begrenzten Beitrag leisten, und *Kernentwickler*, welche die Entwicklung offener Software wie einen Hauptberuf betreiben.[45]

Auch über die Herkunft der Entwickler liegen inzwischen Informationen vor. Robles et al. (2001), Gosh et al. (2002) sowie Lakhani et al. (2002) haben etwa die Nationalität der Befragten ermittelt; die entsprechenden Ergebnisse sind in aggregierter Form in Tabelle 2.7 angegeben. Offensichtlich vollzieht sich die Ent-

[44] Der Gini-Koeffizient entspricht dem Verhältnis der Fläche, die von der Lorenz-Kurve und der Winkelhalbierenden umschlossen wird, zu der Dreiecksfläche unterhalb der Winkelhalbierenden. Er liegt zwischen 0 und 1 und ist eines der gebräuchlichsten Konzentrationsmaße.

[45] Diese Unterscheidung ergibt sich gleichfalls, wenn man statt der bisherigen, inputorientierten Perspektive eine outputorientierte einnimmt. So analysieren Koch und Schneider (2002) über einen Zeitraum von knapp 3 Jahren für das GNOME-Projekt (`www.gnome.org`), wie viele Zeilen Quelltext die einzelnen Entwickler jeweils beigetragen haben. Sie zeigen, daß 17 % der Entwickler 80 % des Quelltexts geschrieben haben. Sehr ähnliche Ergebnisse erhalten Mockus, Fielding und Herbsleb (2000) bei einer Untersuchung des Apache-Projekts sowie Gosh und Prakash (2000).

	Europa	Nordamerika	Rest der Welt
Robles et al. (2001)	55 %	35 %	10 %
Gosh et al. (2002)	73 %	14 %	13 %
Hertel et al. (2003)	37 %	48 %	15 %
Lakhani et al. (2002)	47 %	42 %	11 %

Tabelle 2.7: Nationalität von Entwicklern offener Software

wicklung offener Software primär in der westlichen Welt, also in Nordamerika und in Europa.[46] Dieses liegt in erster Linie sicherlich darin begründet, daß sich derartiges Tun ohne ein gewisses Qualifikations- und Wohlstandsniveau nicht denken läßt. Das auffallende Fehlen japanischer Entwickler mag aber als Indiz für die Bedeutung kultureller Faktoren bei der Entwicklung offene Software gelten.

Im Hinblick auf die im nächsten Kapitel behandelte Frage, warum sich Einzelpersonen an offenen Entwicklungsprojekten beteiligen, ist zudem relevant, daß viele Entwickler nicht aus eigenem Antrieb, sondern ihren Beitrag auf Weisung oder zumindest mit Billigung ihres Arbeitgebers während der Arbeitszeit leisten. So werden 21 % der von Robles et al. (2001) Befragten für die Entwicklung offener Software entlohnt, bei Jørgensen (2001b) sind es sogar 43 %. Ähnliche Ergebnisse erhalten Hertel et al. (2003): In ihrer Untersuchung werden 43 % der als Kernentwickler identifizierten Einzelpersonen zumindest gelegentlich bezahlt; zudem geht mit dieser Bezahlung ein höherer Arbeitseinsatz einher. Genauere Zahlen diesbezüglich geben Hars und Ou (2002). In ihrer Studie werden zwar nur 16 % aller Entwickler bezahlt, diese haben aber 38 % aller Arbeitsstunden geleistet. Bei Lakhani und Wolf (2005) stammen gar 53 % aller Arbeitsstunden von dafür direkt entlohnten Einzelpersonen.

[46] Die signifikanten Abweichungen zwischen den Studien bezüglich der Anteile an Entwicklern aus Nordamerika und Europa erklärt sich wohl teilweise aus dem Ursprung der Befragungen. Die Untersuchungen von Robles et al. (2001), Gosh et al. (2002) und Hertel et al. (2003) wurden an der TU Berlin, der Universität Maastricht bzw. der Universität Kiel initiiert, die von Lakhani et al. (2002) am MIT.

Die Entwicklung offener Software wird also keinesfalls nur von Einzelpersonen, sondern mit gleichem Gewicht auch von Unternehmen getragen. Die Unternehmen, denen diesbezüglich eine besondere Bedeutung zukommt, werden nachfolgend kurz beschrieben.

2.3.1.2 Unternehmen

Zunächst soll das Engagement der 20 laut dem Software Magazine (2004) umsatzstärksten[47] Softwareunternehmen weltweit beleuchtet werden. Unter diesen befinden sich breit aufgestellte IT-Unternehmen wie IBM und Hewlett-Packard, vollkommen auf die Entwicklung von Software fokussierte Unternehmen wie Microsoft und SAP, Beratungsunternehmen wie CSC und Accenture sowie allgemeine Technologiekonzerne wie Lockheed Martin. Recherchen auf den offiziellen Internet-Seiten dieser Unternehmen lassen drei grundsätzliche Reaktionen auf offene Software erkennen (vgl. auch Tabelle 2.8):

- Einige haben offene Software erkennbar in ihre *Strategie* integriert und setzen zum Teil erhebliche Mittel ein. So hat IBM allein im Jahr 2001 über 1 Mrd. $ in Linux investiert und unterstützt über 150 weitere Entwicklungsprojekte in unterschiedlichster Form.[48]

- Weitere Unternehmen unterstützen offene Software hingegen eher *opportunistisch*, also dann, wenn es nur geringen Aufwand erfordert und den eigenen Zielen unmittelbar dient. Beispielsweise versucht Cisco, solche Entwicklungsprojekte, welche die eigenen kommerziellen Produkte ergänzen, durch die Bereitstellung einer Infrastruktur[49] zu fördern und zu bündeln. Eigene Ressourcen setzt Cisco hingegen nur sparsam ein.

[47] Die angegebenen Umsätze umfassen die Geschäfte mit Lizenzen, der Entwicklung von Individualsoftware und softwarebezogenen Dienstleistungen.

[48] Unter `www.ibm.com/developerworks/opensource` werden die Aktivitäten von IBM im Zusammenhang mit offener Software genauer ausgeführt.

[49] Diese Infrastruktur ist unter `cosi-nms.sourceforge.net` erreichbar.

Rang	Unternehmen	Umsatz†	Art und Umfang der Beteiligung‡
1	IBM	56.946	strategisch, sehr viele Projekte
2	Microsoft	32.190	-
3	EDS	20.000	-
4	Lockheed Martin	15.276	-
5	CSC	13.846	-
6	Accenture	13.397	opportunistisch, sehr wenige Projekte
7	Hewlett-Packard	10.165	strategisch, viele Projekte
8	Oracle	9.475	strategisch, viele Projekte
9	Hitachi	9.345	opportunistisch, wenige Projekte
10	SAP	8.779	strategisch, sehr wenige Projekte
11	Capgemini	7.224	-
12	NTT Data Corp	6.943	strategisch, wenige Projekte
13	Unisys	4.692	strategisch, wenige Projekte
14	Ingram Micro	4.520	-
15	SYNNEX	4.126	-
16	ACS	3.787	-
17	Sun Microsystems	3.641	strategisch, sehr viele Projekte
18	Getronics	3.353	-
19	Cisco Systems	3.209	opportunistisch, wenige Projekte
20	Atos Origin	3.035	-
		†Mio. $	‡Subjektive Einschätzung

Tabelle 2.8: Beteiligung der größten Softwareunternehmen an der Entwicklung offener Software

- Schließlich untersützt die Hälfte der betrachteten Unternehmen offene Softwareentwicklung offiziell nicht.

Neben den großen, bereits etablierten Softwareunternehmen sind auch solche Unternehmen zu nennen, die erst in Reaktion auf das Aufkommen offener Software gegründet wurden. Ihre Geschäftsmodelle sind dementsprechend fokussiert und zielen üblicherweise darauf, die Verbreitung offener Software zu fördern, um anschließend vom Verkauf komplementärer Güter und Dienstleistungen zu profitieren.[50] Gegründet wurden diese Unternehmen meist in den

[50]Fink (2003) beschreibt dieses Geschäftsmodell und seine Varianten ausführlich.

90er Jahren, finanziert werden sie mit Wagniskapital, welches von spezialisierten Beteiligungsgesellschaften oder vermögenden Privatpersonen, sogenannten "Business Angels", bereitgestellt wird, und auf Aktienmärkten mit niedrigen Zulassungshürden, beispielsweise der amerikanischen NASDAQ. Die Umsätze überschreiten selten 100 Mio. $ und sind häufig sogar so gering, daß die dauerhafte Existenz vieler Unternehmen kaum gesichert ist. Es erscheint daher weder sinnvoll noch möglich, hier sämtliche Unternehmen dieser Kategorie aufzuführen; vielmehr werden nachfolgend zwei repräsentative kurz vorgestellt.

Die bekannteste Firma mit einem auf offene Software gegründeten Geschäftsmodell ist wohl Red Hat.[51] Die Grundlage dieses Unternehmens bildet die Distribution "Red Hat Linux", in welcher der bereits beschriebene Linux-Kernel mit weiteren Programmen zu einem aus Konsumentensicht vollständigen Betriebssystem gebündelt wird.[52] Die Distribution selbst ist zwar noch immer kostenlos erhältlich, verschafft Red Hat aber einen starken Markennamen, mittels dessen das Unternehmen am Verkauf von Handbüchern, technischer Unterstützung und umfassender, auf Linux basierender Problemlösungen für Unternehmen verdienen kann. Red Hat wurde im Jahr 1993 gegründet und hat im Geschäftsjahr 2004 mit ca. 740 Mitarbeitern einen Umsatz von 125 Mio. $ erwirtschaftet.[53] Seit 1999 ist das Unternehmen an der NASDAQ notiert und hatte im Mai 2005 eine Marktkapitalisierung von über 2 Mrd. $. Der schärfste Konkurrent, das deutsche Unternehmen SuSE, wurde Anfang 2005 von Novell, mit 1.105 Mio. $ auf Platz 42 der Rangliste des Software Magazine (2004), übernommen.

Ein allein auf offene Software gegründetes Unternehmen neuerer Prägung ist MySQL, insbesondere bekannt durch seine gleichnamige Informationsmanage-

[51] Eine ausführliche Darstellung dieses Unternehmens und seines Geschäftsmodells gibt Young (1999), einer seiner Gründer.

[52] Während Betriebssysteme aus technischer Sicht recht eng abgegrenzte Aufgaben wahrnehmen, erwarten Konsumenten heutzutage einen vielfältigen Funktionsumfang.

[53] Unter `phx.corporate-ir.net/phoenix.zhtml?c=67156&p=irol-reportsannual` liegt der Jahresabschluß Red Hats für das Geschäftsjahr 2004 vor.

mentsoftware.[54] Anders als Red Hat im Fall von Linux trägt MySQL die Entwicklung seines Kernprodukts im wesentlichen selbst; von der Öffnung seiner Software verspricht sich das Unternehmen im wesentlichen die Unterstützung der Entwicklergemeinschaft beim Testen und der Bereinigung von Fehlern sowie eine lebendige Peripherie, in der das allein wenig nützliche MySQL an größere Systeme angebunden wird. Aufgrund des hohen Eigenanteils bei der Entwicklung kann MySQL nicht nur vom Verkauf komplementärer Produkte und Dienstleistungen, sondern auch von dem sogenannter *dualer Lizenzen* profitieren. Dabei stellt MySQL als Urheber den Quelltext sowohl unter der GPL als auch unter einer proprietären Lizenz bereit. Für die letztgenannte Lizenz sind zwar Gebühren zu entrichten, dafür erlaubt sie es dem Lizenznehmer, die Software auf der Quelltextebene in ein proprietäres Produkt zu integrieren. MySQL beschäftigte im Jahr 2004 mehr als 170 Mitarbeiter; Umsatz und Gewinn sind unbekannt,[55] da sich das Unternehmen in Privatbesitz befindet. Seit dem Jahr 2001 wird es von mehreren Wagniskapitalgebern finanziert, namentlich den Unternehmen "Benchmark Capital" und "Index Ventures" sowie diversen Business Angels. Im Jahr 2004 hat das deutsche Unternehmen SAP Teile seiner Informationsmanagementsoftware an MySQL abgetreten, wo diese unter der Bezeichnung MaxDB als offene Software weiterentwickelt werden.

Einen etwas anderen Blickwinkel auf das Wirken von Unternehmen für die Entwicklung offener Software nimmt Henkel (2004c) ein. Er beschränkt sich mit einer empirischen Untersuchung auf die Entwicklung von Embedded-Linux, einer besonders schlanken Variante des Linux-Kernels, welche für den Endbenutzer völlig unsichtbar in elektronischen Geräten wie etwa Mobiltelefonen, Digitalfernsehern und Waschmaschinen eingesetzt wird. Der Vorteil dieser verengten Perspektive liegt darin, daß alle relevanten Unternehmen recht einfach zu identifizieren sind und daß zudem Einzelpersonen als Entwickler keine nennenswerte Rolle spielen; es sind somit zwar genaue Aussagen möglich, gleichzeitig man-

[54] Genaue Informationen zu MySQL finden sich unter `www.mysql.com`.

[55] Eine unzureichend anonymisierte Fallstudie Dahlanders (2004) legt nahe, daß MySQL im Jahr 2002 etwa 5,3 Mio. € Umsatz und 1,6 Mio. € Verlust erwirtschaftet hat.

gelt es ihnen aber an Allgemeingültigkeit. Für Embedded-Linux gilt nun laut Henkel (2004c), daß 49 % der in Unternehmen verfaßten Quelltexte der Entwicklergemeinschaft aktiv zugänglich gemacht werden.[56] Des weiteren wird die Entscheidung zur Veröffentlichung fast immer bewußt getroffen, da sie rund 80 % der untersuchten Unternehmen auf offizielle, vom Management definierte Richtlinien gründen.[57]

2.3.2 Nutzer

Selbstverständlich verwenden die meisten Nutzer offene Software in der gleichen Weise wie proprietäre allein als Endprodukt und verzichten darauf, sich am Entwicklungsprojekt zu beteiligen oder den Quelltext einzusehen (vgl. Hissam und Weinstock, 2001). Einige jedoch liefern im Rahmen ihrer Möglichkeiten freiwillig einen Beitrag, etwa indem sie Handbücher und Anleitungen verfassen oder als Tester Programmfehler dokumentieren und Lösungsvorschläge unterbreiten.[58]

Gerade als Tester spielen Nutzer laut Raymond (2001) in vielen Entwicklungsprojekten eine wichtige Rolle. Dieses läßt sich etwa am Beispiel der offenen Software “fetchmail” darstellen, einem kleinen, von Eric S. Raymond geschriebenen

[56] Eine Veröffentlichungsquote von 49 % mag zwar zunächst als hoch erscheinen, ist aber tatsächlich überraschend gering, da ja der Linux-Kernel und somit auch alle seine Erweiterungen unter der GPL stehen, welche die Weitergabe einer Software ohne die dazugehörigen Quelltexte verbietet. Die genannte Veröffentlichungsquote beschreibt daher den Anteil der Quelltexte, die freiwillig und aktiv preisgegeben werden. Die übrigen Quelltexte werden hingegen im Rahmen des lizenzrechtlich Erlaubten bestmöglich geheimgehalten, etwa indem ihr Entwickler sie nur an seine Kunden weitergibt, die ihrerseits kein Interesse an der Veröffentlichung haben. Im Fall von Embedded-Linux besteht zudem die Möglichkeit, daß ein Hardwarehersteller seine Quelltextmodifikationen erst dann veröffentlicht, wenn er das dazugehörige elektronische Gerät auf den Markt bringt. Da nun die Entwicklung eines elekronischen Geräts regelmäßig mehr Zeit erfordert als sein Lebenszyklus im Markt, ermöglicht dieses Vorgehen eine sehr exklusive Nutzung der eigenen Modifikationen, die in gewisser Hinsicht einer Nichtveröffentlichung gleichkommt.

[57] Eine genaue Beschreibung der zugrundeliegenden Daten geben Henkel und Tins (2004).

[58] Mit der offenen Software Bugzilla existiert sogar ein System, welches die Einbindung von Nutzern in die Fehlerbereinigung informationstechnisch unterstützt.

```
fetchmail-6.2.5 (Wed Oct 15 18:39:22 EDT 2003), 23079 lines:

* Updated Spanish, Turkish, and German translation files.
* Matthew Gregan's patch to handle garbage lengths from dbmail;
  closes Debian bug #207919.
* Fix IMAP query so new-message count doesn't include deleted messages.
* Man page typo fix, closes Debian bug #205892.
* OpenSSL cleanup patches from levinedl@acm.org.
* Benjamin Drieu's patch to fix Debian bug #212240, no oversized-message
  flushing if both "flush" and "limit" were specified.
* Benjamin Drieu's patch for Debian bug #156592, incorrect handing of host/port
  option.
* Smash all NULs out of headers right after the socket read.
* Dup-killer code now keys on an MD5 hash of the raw headers.
* Sunil Shetye's patches to break up fetching of sizes and UIDLs.

  There are 599 people on fetchmail-friends and 748 on fetchmail-announce.
```

Abbildung 2.3: Auszug aus der Datei "NEWS" von fetchmail 6.2.5

Programm, welches für Endanwender unsichtbar den Transport von E-Mails besorgt und sich somit nur an technisch versierte Nutzer richtet.[59] Abbildung 2.3 zeigt einen Auszug aus der Datei "NEWS", welche gemeinsam mit dem Quelltext verbreitet wird und wichtige Programmneuerungen benennt. Dieser Auszug illustriert dreierlei: Erstens ist fetchmail mit 23.079 Zeilen Quelltext ein kleines Programm, welches in der Tat mühelos von einer Einzelperson entwickelt werden kann. Zweitens wird diese Einzelperson von 599 Abonnenten der Mailing-Liste "fetchmail-friends", die sich unter anderem der Diskussion von Fehlern widmen, unterstützt. Und drittens mündet diese Unterstützung auch in verwertbare Beiträge, da fünf von zehn der nennenswerten Neuerungen in der aktuellen Version 6.2.5 von Nutzern eingereichte Patches sind, kleine Quelltextänderungen, die Fehler ausmerzen. Folglich ermöglicht also die Offenlegung des Quelltextes eine Einbindung der Nutzer in den Entwicklungsprozeß, die Anbietern proprietärer Software in dieser Form verschlossen ist.

Lakhani und von Hippel (2003) untersuchen zudem anhand des Apache-Projekts, wie Nutzer sich untereinander helfen, um das Fehlen einer für proprietäre Software typischerweise vom Hersteller bereitgestellten Anwendungsunterstützung zu kompensieren. Sie zeigen, daß die in elektronischen Nachrichtenforen organisierte wechselseitige Hilfe insgesamt gut funktioniert. Als Grund nennen sie,

[59] Unter `www.catb.org/~esr/fetchmail/` finden sich weitere Angaben zu fetchmail.

daß die Forenbesucher 99 % der Gesamtzeit damit verbringen, die Beiträge anderer zu lesen um zu lernen. Nur 1 % ihrer Zeit wenden sie dementsprechend dafür auf, anderen zu helfen; zudem benantworten sie nur solche Fragen, deren Antwort ihnen sofort bekannt ist. Die individuellen Kosten für das Hilfesystem Apaches sind also sehr gering; effektiv wird es durch eine hohe Besucherzahl, effizient durch eine informationstechnische Umsetzung mit guten Such- und Klassifizierungsfunktionen.[60]

2.3.3 Dachorganisationen

2.3.3.1 Projektspezifische Organisationen

Gerade große offene Entwicklungsprojekte werden, wie in Abschnitt 2.2.2 bereits gesehen, oft von projektspezifischen Dachorganisationen überspannt. Diese sind im Regelfall als gemeinnützige Stiftungen oder Vereine organisiert und nehmen eine Reihe von Aufgaben wahr, welche die eigentliche Entwicklungsarbeit unterstützen und ergänzen. Typische Aufgabenschwerpunkte sind[61]

- die *organisatorische* Unterstützung, insbesondere die Bereitstellung von Infrastruktur und formalen Prozessen,
- die *rechtliche* Vertretung des Entwicklungsprojekts nach außen, insbesondere die Sicherung von Markenrechten und der Schutz einzelner Entwickler vor Rechtsstreitigkeiten,
- sowie die *finanzielle* Unterstützung, insbesondere die Entgegennahme von Spenden und deren Nutzung gemäß der Ziele des Entwicklungsprojekts.

[60] Ein vergleichbares Ergebnis erhalten Constant, Sproull und Kiesler (1996) bei der Untersuchung eines firmeninternen, aber auf Freiwilligkeit basierenden Hilfesystems, welches denen offener Entwicklungsprojekte strukturell sehr ähnlich ist.

[61] Wichtige projektspezifische Organisationen sind die Apache Software Foundation, die Eclipse Foundation, die Mozilla Foundation, die Python Software Foundation und die Perl Foundation. Ausführliche Darstellungen ihrer jeweiligen Ziele finden sich unter `www.apache.org`, `www.eclipse.org`, `www.mozilla.org`, `www.python.org` bzw. `www.perl.org`.

Projektspezifische Dachorganisationen übernehmen also ganz gezielt jene Funktionen, die ein informeller Zusammenschluß von Einzelnen nur unzureichend zu tragen vermag.[62]

2.3.3.2 Lobbyorganisationen

Des weiteren existieren im Umfeld offener Software mit der OSI und der FSF zwei Lobbyorganisationen, welche sich nicht nur um einzelne Projekte, sondern um offene Software insgesamt bemühen.

Die FSF ist allerdings keine reine Lobby. Sie fungiert zudem als Dachorganisation des in Abschnitt 2.2.1 beschriebenen GNU-Projekts und arbeitet in dieser Funktion wie eben dargestellt. In ihrer Funktion als Lobby für freie Software[63] konzentriert sie sich hingegen auf die soziale Bedeutung dieses Konzepts; insbesondere propagiert sie freie Software als informationstechnisches Äquivalent zu einer freien Gesellschaft (vgl. Stallman, 1999). Des weiteren gewährt die FSF juristischen Beistand bei Verstößen gegen die GPL, pflegt gemeinsam mit der UNESCO ein Verzeichnis freier Software und bietet die Plattform Savannah als kostenlos verfügbare Infrastruktur für offene Entwicklungsprojekte. Die für ihre Arbeit erforderlichen Mittel bezieht die FSF zu fast 70 % aus Spenden von Privatpersonen; weitere Mittel stammen von Sponsorunternehmen wie etwa Google, Hewlett-Packard, IBM und Sun sowie aus dem Vertrieb freier Software.[64]

Die OSI widmet sich hingegen ausschließlich der Lobbyarbeit. Ihr Ziel ist im Kern die Stärkung des Konzepts offener Software innerhalb der Softwareindustrie. Dafür stützt sich die OSI im wesentlichen auf einen prominent besetzten

[62] Kleinere offene Softwareprojekte verfügen meistens über keine formale Dachorganisation. Als Infrastruktur können sie etwa die Technologiedemonstration SourceForge.net oder die von der FSF unter `savannah.nongnu.org` bereitgestellte Plattform Savannah nutzen.

[63] Die FSF zieht die Bezeichnung "Free Software" der inzwischen gebräuchlicheren "Open Source Software" vor und macht mit ihrem Wahlspruch "Free as in 'free speech', not as in 'free beer'!" sehr deutlich, wie sie diese Bezeichnung verstanden wissen will.

[64] Die FSF stellt sich unter `www.fsf.org` umfassend vor.

Vorstand. Alle seine Mitglieder sind zum einen tief in der Hacker-Kultur verwurzelt, zum anderen aber auch seit langem in herausgehobenen Positionen in der Industrie tätig.[65] Des weiteren verleiht die OSI solchen Lizenzen, welche mit der OSD konform sind, das geschützte Zertifizierungszeichen "OSI Certified".[66] Insofern kann sie bis zu einem gewissen Grad kontrollieren, welche Software als "Open Source Software" firmieren darf.

2.4 Offene Software aus Entwicklersicht

2.4.1 Entwicklungsprozeß

Kommerzielle Softwareentwicklung gilt allgemein als ein schwer steuerbares Unterfangen. Die Allgegenwart überzogener Fristen und gesprengter Budgets hat langjährige Beobachter gar zu der Einschätzung veranlaßt, die Industrie befinde sich in einer sogenannten Software-Krise. Als Hauptursache dieser Krise werden die eingesetzten Entwicklungsprozesse angesehen. Insbesondere das einflußreiche Wasserfallmodell steht in der Kritik, da es einen sehr starren Prozeß vorschreibt: Auf die *Analyse* des Problems sollen der *Entwurf* einer Lösung sowie die *Implementierung* der Lösung in Quelltext folgen; abschließend sind *Tests* durchzuführen und Fehler zu beheben.[67] Dieses sequentielle Vorgehen führt insbesondere dazu, daß Fehler erst äußerst spät erkannt werden und dementsprechend schwer zu beheben sind. Daher sucht die Softwareindustrie intensiv nach

[65] Der Vorstand der OSI wird unter `opensource.org/docs/board.php` vorgestellt. Im Januar 2005 arbeiteten drei seiner fünf Mitglieder bei großen Unternehmen mit einem strategischen Interesse an offener Software, namentlich Ken Coar bei IBM, Danese Cooper bei Sun und Michael Tiemann bei Red Hat.

[66] Unter `opensource.org/docs/certification_mark.php` finden sich genaue Angaben zur Zertifizierung.

[67] Die letzte Phase, in der die Software getestet, um Fehler bereinigt und gewartet wird, verursacht üblicherweise die höchsten Kosten. Laut Cusumano (1991) fallen sogar mehr als 80 % aller Kosten erst nach der Implementierung an.

alternativen Prozessen, welche die beschriebenen Probleme lindern können.[68] Eine solche Alternative ist die innerhalb der meisten offenen Softwareprojekte praktizierte Basar-Methode. Ihre wesentlichen Charakteristika werden nachfolgend in Anlehnung an Mockus, Fielding und Herbsleb (2000), Jørgensen (2001a) und Edwards (2003) vorgestellt.

Im Zentrum der Basar-Methode steht der *Quelltext*, und zwar in doppeltem Sinn. Zum einen beschreibt er jederzeit den aktuellen Entwicklungsstand der Software. Zum anderen steht die Arbeit am Quelltext auch immer am Anfang einer jeden Veränderung oder Erweiterung der Software. Anders als im Wasserfallmodell wird also auf eine formale Analyse- und Entwurfsphase gänzlich verzichtet und sofort mit der Implementierung begonnen. Anschließend wird der neu geschriebene Quelltext zunächst veröffentlicht und *begutachtet*. Auf der Grundlage dieser Begutachtung wählt ein hochrangiger, mit den entsprechenden Rechten ausgestatteter Entwickler den Quelltext gegebenenfalls für die Aufnahme in die sogenannte *Entwicklerversion* der Software aus. Dabei hat er unbedingt die Lauffähigkeit dieser Version sicherzustellen, damit sie parallel von möglichst vielen Projektmitgliedern getestet und *um Fehler bereinigt* werden kann. Erst danach wird der Quelltext auch in die für den alltäglichen Einsatz geeignete *stabile Version* übernommen. Werden hingegen im Verlauf Mängel offenbar, so ist der Quelltext zu überarbeiten und der gesamte Prozeß beginnt erneut. Offensichtlich ist die Basar-Methode ursprünglich aus dem Bemühen entstanden, den Quelltext eines offenen Softwareprojekts vor unerwünschten Änderungen zu schützen.

Wie auch in Abbildung 2.4 dargestellt, sind die verschiedenen Aktivitäten innerhalb der Basar-Methode nicht linear, sondern zyklisch angeordnet. Jede einzelne Quelltextveränderung durchläuft diesen Zyklus grundsätzlich nach dem "Release, when ready"-Prinzip, welches besagt, daß externe Faktoren wie etwa Fristen für die Entscheidung, ob die Veränderung auf ihrem Weg in die stabile Version

[68] Das Wasserfallmodell geht auf Royce (1970) zurück; wichtige alternative Prozeßmodelle werden von Boehm (1988), Jacobson, Booch und Rumbaugh (1999) sowie Jeffries et al. (2000) beschrieben.

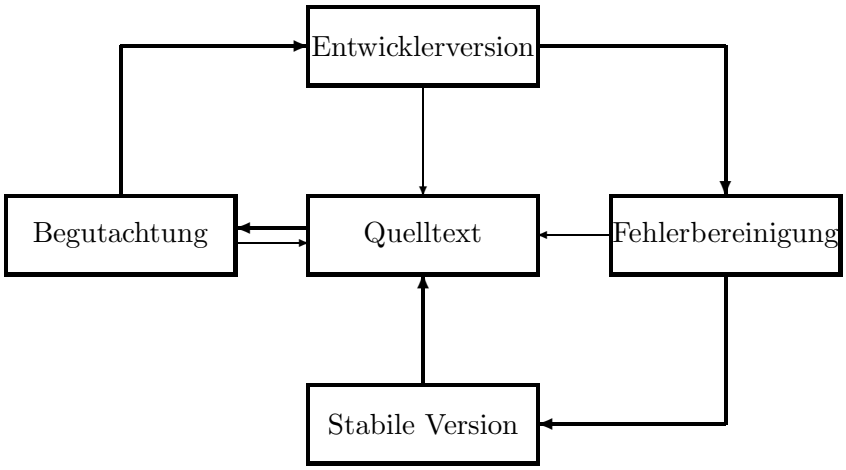


Abbildung 2.4: Softwareentwicklung nach der Basar-Methode

die nächste Stufe erklimmt, keine Rolle spielen dürfen. Da aber üblicherweise an vielen Stellen des Quelltextes gleichzeitig gearbeitet wird, gilt insgesamt das "Release early, release often"-Prinzip, nach dem auch kleine Verbesserungen die Veröffentlichung einer neuen Version der Software rechtfertigen.

Eine Prozeßinnovation stellt die Basar-Methode deshalb dar, weil sie Softwareentwicklung erstmalig konsequent als einen inkrementellen Verbesserungsprozeß begreift, welcher der durchgängigen Qualitätskontrolle bedarf. So vermeidet sie die Schwäche des Wasserfallmodells, Qualitätsprobleme erst sehr spät erkennen zu können. Selbstverständlich hat die Bazar-Methode auch Nachteile. Insbesondere gilt sie laut McConnell (1999) aufgrund des Fehlens einer formalen Analyse- und Entwurfsphase als untauglich, wenn wahrhaft neuartige Software mit einer noch unklaren Architektur entwickelt werden soll.

2.4.2 Führung und Entscheidungsgewalt

Die Basar-Methode bedarf, soll sie nicht ins Chaos führen, zweier Ordnungskräfte: Führung und Entscheidungsgewalt. Die Rolle der Führung ist es, den Bemühungen der Projektmitglieder um Verbesserungen des Quelltextes eine gemeinsame Richtung zu geben. Sie wird laut Lerner und Tirole (2002) im

Regelfall von einem Projektgründer wahrgenommen, welcher aufgrund des Fehlens jeglicher Sanktionsmechanismen über Überzeugungskraft und Charisma im Sinne Max Webers verfügen sollte. Des weiteren ist eine Gewalt erforderlich, welche über die Aufnahme von Quelltext in die offiziellen Versionen der Software befindet. Diese Gewalt kann grundsätzlich eine von drei möglichen Formen annehmen (vgl. Fink, 2003):

- Innerhalb *monarchisch* organisierter Projekte geht alle Entscheidungsgewalt prinzipiell von einer Person aus. Gerade bei größeren Projekten wird zudem oft eine Art Lehenswesen etabliert, bei dem der Monarch seine Gewalt über Teile des Projekts widerruflich abtritt.[69] Ein Beispiel für ein derartiges Projekt ist Linux.

- In *aristokratischen* Projekten entscheidet eine wohldefinierte Gruppe im Konsens. Des weiteren erhebt sie nach eigenem Ermessen weitere Projektmitglieder in den Adelsstand. Ein Beispiel gibt das Apache-Projekt.

- Schließlich existieren noch *demokratische* Projekte, in welchen die Entscheidungsgewalt mittels Wahlen auf Zeit übertragen wird. Ein Beispiel ist OpenOffice.org.

Zudem sind einige offene Softwareprojekte wie etwa Eclipse nicht vollkommen souverän, sondern werden zum Teil etwa von einem Unternehmen von außen kontrolliert. Gerade in derartigen Fällen verschärft sich aber das Problem, daß sich Führung und Entscheidungsgewalt vor den freiwilligen Projektmitgliedern stets erst legitimieren müssen, ganz erheblich.

Kogut und Metiu (2001) weisen zudem auf den Zusammenhang zwischen der Entscheidungsstruktur und der Lizenzierungsart eines offenen Softwareprojekts hin. Danach überdauern in Projekten, welche eine Copyleft-Lizenz verwenden, oftmals monarchische Strukturen, da die Lizenz zumindest davor schützt, daß

[69] Raymond (2001) beschreibt am Beispiel des FetchMail-Projekts, wie sich in monarchischen Projekten die Regierungsübergabe vollzieht.

unzufriedene Splittergruppen den Quelltext in einem geschlossenen Prozeß weiterentwickeln. Projekte mit einer Non-Copyleft-Lizenz müssen sich, da sie über keinen derartigen Schutz verfügen, hingegen weit stärker um Konsens bemühen; sie verfolgen daher eher aristokratische oder sogar demokratische Ansätze.

2.4.3 Modulare Softwarearchitektur und Koordination

Software wird heutzutage üblicherweise nicht als Monolith, sondern als Ensemble zahlreicher wohldefinierter Bausteine entworfen. Die einzelnen Bausteine, nachfolgend als Module bezeichnet,[70] beinhalten jeweils nur einen Teil der gesamten Funktionen der Software. Sie interagieren über exakt spezifizierte Schnittstellen, welche das "Was" des Moduls beschreiben; das "Wie" ist hingegen im Modulkörper verborgen. Im wesentlichen soll eine solche modulare Softwarearchitektur die Software intellektuell beherrschbar machen und ihre arbeitsteilige Entwicklung erleichtern.

Auch offene Software wird meist modular entwickelt. So ist etwa die Architektur von Linux seiner Entstehungsgeschichte entsprechend eng an Unix angelehnt. Dessen Aufbau wiederum beschreiben Kernighan und Pike (1984, S. viii) wie folgt:

> [...] at its heart is the idea that the power of a system comes more from the relationships among programs than from the programs themselves. Many UNIX programs do quite trivial tasks in isolation, but, combined with other programs, become general and useful tools.

Feller und Fitzgerald (2002) untersuchen den Aufbau der offenen Softwareprogramme Apache und Perl und stellen eine sehr ausgeprägte Modularisierung fest. Robles (2004) sieht gar in der *höchstmöglichen* Modularisierung ein gemeinsames Leitbild aller offenen Entwicklungsprojekte.

[70] Tatsächlich existieren neben Modulen noch andere Arten von Bausteinen wie etwa Komponenten, Bibliotheken, Klassen, Programme und Prozeduren. Ihre Unterschiede sind allerdings im Rahmen dieser Arbeit belanglos.

Ursache für dieses Leitbild sind wohl die erheblichen Vorteile, welche modulare Architekturen bei der Entwicklung offener Software bieten: Erstens erleichtern sie die *Koordination* innerhalb der Entwicklergemeinschaft; sobald die einzelnen Module und ihre Schnittstellen definiert sind, ist der Kommunikationsbedarf zwischen Entwicklern, die an verschiedenen Modulen arbeiten, nur noch gering (vgl. Bonaccorsi und Rossi, 2003). Zweitens fördern sie die *Wiederverwendung* offengelegten Quelltexts in anderen Zusammenhängen und reduzieren somit die Entwicklungskosten offener Software insgesamt (vgl. Raymond, 2001). Und drittens ermöglichen sie die *Parallelisierung* der Entwicklungsbemühungen, insbesondere während der Fehlerbereinigung (vgl. Feller und Fitzgerald, 2002).[71]

Das genaue Verständnis modularer Softwarearchitekturen hilft auch, die brennende Frage zu beantworten, wie die große Anzahl von Beteiligten[72] an umfangreichen offenen Entwicklungsprojekten ihre Anstrengungen koordiniert und ein funktionierendes Endprodukt erschafft. Tatsächlich ist die Zahl derer, die an ein und demselben Modul arbeiten, sehr gering. Die hohe Gesamtzahl entsteht erst durch die Aggregation der Module. Des weiteren existieren aber auch viele kleine, kaum bekannte offene Softwareprojekte, an denen sich entsprechend ihres Umfangs nur wenige Entwickler beteiligen. So untersucht Krishnamurthy (2002) 100 Projekte, welche aufgrund ihrer geringen Größe keine eigene Infrastruktur benötigen, sondern sich der von SourceForge.net bedienen. Er stellt fest, daß durchschnittlich vier Entwickler an einem Projekt arbeiten; der Median beträgt sogar nur Eins. Gosh, Robles und Glott (2002) kommen bei einer Untersuchung von rund 15.000 Entwicklungsprojekten zu vergleichbaren Ergebnissen.

[71] Die starke Parallelisierung während der Fehlerbereinigung, wie sie in fast allen offenen Entwicklungsprojekten praktiziert wird, gilt als Schlüssel zu hochwertiger Software. Ihr liegt der innerhalb der Hacker-Kultur verbreitete Lehrsatz "Given enough eyeballs, all bugs are shallow", zu Ehren von Linus Torvalds auch als "Linus' Law" bezeichnet, zugrunde (vgl. Raymond, 2001).

[72] Für große Entwicklungsprojekte ist es im allgemeinen schwer, die Anzahl der beteiligten Entwickler auch nur annähernd zu bestimmen. Im Fall des Linux-Kernels etwa reichen die Schätzungen von 1.200 (McConnell, 1999) über 40.000 (Raymond, 2001) bis hin zu mehreren Hunderttausend (Torvalds und Diamond, 2001).

2.4.4 Entwicklungswerkzeuge

Softwareentwicklung erfordert wiederum spezielle Software, sogenannte Entwicklungswerkzeuge. Gerade die Entwicklung dieser speziellen Software ist traditionell eine Domäne der Hacker-Kultur. So hat die FSF bereits in den 80er Jahren mit dem Compiler gcc und dem Quelltexteditor Emacs grundlegende Entwicklungswerkzeuge bereitgestellt.[73] Und von allen auf SourceForge.net beheimateten offenen Softwareprojekten zielten im Januar 2004 rund 34 % auf die Bedürfnisse von Entwicklern (vgl. Gutsche, 2004). Zudem existieren einige offene Werkzeuge, welche gezielt die Entwicklung gemäß der Basar-Methode unterstützen; bekannte Beispiele sind das Fehlerverwaltungssystem Bugzilla und das Quelltextverwaltungssystem CVS, welches Quelltexte, an denen mehrere Entwickler parallel arbeiten, vollautomatisch vor Inkonsistenzen schützt.[74]

Aufgrund des breiten Angebots an offenen Werkzeugen kann sich offene Softwareentwicklung weitgehend unabhängig von proprietärer Software vollziehen. Diese Unabhängigkeit wird auch ausdrücklich angestebt. Beispielsweise konstatiert Matthias Ettrich (2004, S. 186), ein prominenter Entwickler offener Software:

> Das Einsetzen nicht-freier Werkzeuge im Entwicklungsprozess wird allgemein als Makel empfunden und nur in Ausnahmefällen toleriert, bei denen keine langfristigen Abhängigkeiten geschaffen werden.

Eine Untersuchung der von Projekten auf SourceForge.net verwendeten Programmiersprachen stützt diese Aussage. So wurden im Februar 2005 für deutlich über 90 % aller Projekte solche Sprachen verwendet, für die offene Entwicklungsumgebungen gut verfügbar sind.[75]

[73] Genaue Angaben zu gcc und Emacs werden unter `www.gnu.org` bereitgestellt.

[74] Unter `www.bugzilla.org` und `www.cvshome.org` finden sich genaue Beschreibungen von Bugzilla und CVS.

[75] Die am häufigsten eingesetzten Sprachen mit offenen Entwicklungsumgebungen sind C, C#, C++, Java, JavaScript, Perl, PHP, SQL, Python und Unix-Skriptsprachen, die mit proprietären sind Assembler, Delphi und Visual Basic.

Im Regelfall liegen also nicht nur die Quelltexte offen, sondern auch die für ihre Verarbeitung erforderlichen Werkzeuge. Das Recht auf Selbstbestimmung, welches offene Software ihren Nutzern grundsätzlich einräumt, reicht folglich deutlich weiter, als es eine reine Betrachtung offener Lizenzbestimmungen vermuten ließe. Des weiteren reduziert die Verwendung offener Entwicklungswerkzeuge offensichtlich die Entwicklungskosten.

2.5 Offene Software aus Nutzersicht

2.5.1 Angebot

Das Angebot an offener Software ist zweifelsfrei umfangreich. Zwar ist es schwierig, die Anzahl von offenen Softwareprojekten genau zu bestimmen; da aber im Februar 2005 auf SourceForge.net knapp 100.000 Projekte registriert waren, von denen rund 44 % mindestens im sogenannten Beta-Stadium, also grundsätzlich für die Verwendung durch Endnutzer geeignet waren, kann als untere Grenze ein Angebot von 40.000 verschiedenen offenen Softwareprogrammen gelten. Faktisch wird diese Zahl aufgrund der Tatsache, daß nicht alle Entwicklungsprojekte auf die Infrastruktur von SourceForge.net zurückgreifen, natürlich noch größer sein.

Neben dem Umfang des Angebots ist des weiteren auch seine Struktur interessant. Diese wird unter Verwendung der in Abschnitt 2.2.2 eingeführten Kategorien in Tabelle 2.9 beschrieben. Alle Angaben fußen dabei auf einer Sichtung der im Februar 2005 in den Verzeichnissen FreshMeat, SourceForge und SourceWell katalogisierten sowie der in Red Hat Linux 6.0 enthaltenen offenen Software.[76] Demnach liegt der Schwerpunkt des Angebots auf Anwendun-

[76] Die drei für diese Analyse verwendeten Verzeichnisse finden sich unter `www.freshmeat.net`, `www.sourceforge.net` und `sourcewell.berlios.de`, Angaben zu Red Hat Linux unter `www.redhat.com`. Das als Vergleichsmaßstab dienende proprietäre Angebot entspricht dem Katalogangebot des Händlers SoftwareHouse unter `www.softwarehouse.de`. Die Zuordnung der jeweiligen Originalkategorien zu den hier verwendeten wird in Anhang B beschrieben.

Kategorie	FreshMeat	Red Hat	SourceForge	SourceWell	Gesamt	Proprietär
System	22,0 %	40,3 %	18,6 %	18,8 %	19,9 %	5,6 %
Netzwerkmanagement	24,3 %	0,0 %	28,0 %	28,5 %	26,5 %	13,6 %
Informationsmanagement	4,8 %	0,0 %	4,4 %	1,7 %	4,5 %	3,5 %
Anwendungen	32,1 %	32,9 %	34,7 %	31,2 %	33,7 %	67,3 %
Programmentwicklung	16,1 %	20,8 %	12,4 %	15,9 %	13,8 %	4,7 %
Sonstiges	0,8 %	6,1 %	1,9 %	4,0 %	1,5 %	5,3 %
Gesamt	36,5 %	0,3 %	61,6 %	1,6 %	100,0 %	100,0 %
Absolut	62.208	578	104.971	2.727	170.484	951

Tabelle 2.9: Struktur des Angebots an offener Software

gen, gefolgt von Netzwerkmanagement-, System-, Programmentwicklungs- und schließlich Informationsmanagementsoftware (vgl. vorletzte Spalte von Tabelle 2.9).[77] Das Besondere dieser Angebotsstruktur wird allerdings erst offenbar, vergleicht man sie mit der proprietärer Software (vgl. letzte Spalte von Tabelle 2.9). Dann zeigt sich, daß überproportional viel offene Software auf die technischen Kategorien System-, Netzwerkmanagement-, Programmentwicklungs- und Informationsmanagementsoftware entfällt, wohingegen auch für technisch unversierte Nutzer geeignete Anwendungen unterproportional häufig entwickelt werden. Die Ursache für diese technikerlastige Angebotsstruktur ist, daß offene Software, wie im nächsten Kapitel ausführlich dargelegt wird, oft für den Eigenbedarf entwickelt wird.

Abgerundet wird dieses Bild von Gosh et al. (2002), die rund 2.800 Entwickler offener Software nach ihrem Arbeitsschwerpunkt befragten. Die in Tabelle 2.10 angegebenen Ergebnisse dieser Studie decken sich für Netzwerkmanagementsoftware und Anwendungen mit denen aus Tabelle 2.9. Des weiteren legen sie nahe, daß die angebotenen Anwendungen den Büro- und den Heimbedarf in gleichem Maße bedienen. Die anderen in Tabelle 2.10 verwendeten Kategorien können aufgrund lückenhafter Angaben den hier verwendeten leider nicht zugeordnet werden.

[77] Man beachte, daß sich die jeweils von FreshMeat, SourceForge und SourceWell bereitgestellten Angebote strukturell kaum unterscheiden. Lediglich das von Red Hat weicht deutlich ab, vermutlich aber im wesentlichen deshalb, weil die verwendeten Originalkategorien keine Unterscheidung von System- und Netzwerkmanagementsoftware erlauben.

Kategorie	Anteil
Netzwerkmanagement	26,9 %
Webdienste	22,7 %
Büroanwendungen	17,8 %
Heimanwendungen	16,4 %
Sonstiges	16,3 %

Tabelle 2.10: Arbeitsschwerpunkte offener Softwareentwickler

2.5.2 Motive für die Nutzung

Die Entscheidung für oder gegen den Einsatz einer bestimmten Software wird laut Brügge et al. (2004) üblicherweise auf der Grundlage dreier Arten von Kriterien gefällt:

- *Operative* Kriterien erfassen, inwieweit die betrachtete Software den gegenwärtigen Qualitätsanforderungen genügt. Je nach Einsatzgebiet sind dabei die Qualitätsdimensionen Zuverlässigkeit, Geschwindigkeit, Sicherheit, Skalierbarkeit[78] und Benutzerfreundlichkeit mehr oder weniger entscheidungsrelevant.

- *Strategische* Kriterien blicken hingegen in die Zukunft, indem sie ihr Augenmerk auf Aspekte wie die Vermeidung von Lock-In auf einen Hersteller, den Einfluß auf die Weiterentwicklung der Software sowie ihre allgemeine Zukunftssicherheit legen.

- Schließlich beleuchten *monetäre* Kriterien die Kostenseite. Gerade im Kontext dieser Arbeit ist es dabei wichtig anzumerken, daß die Kosten über den gesamten Nutzungszeitraum einer Software nicht nur für Lizenzen, sondern auch für erforderliche Infrastruktur, Schulungen, Einführungs- und Anpassungsarbeiten sowie Wartung und Updates anfallen (vgl. Mer-

[78] Eine Software skaliert dann gut, wenn sie auch bei einem wachsenden Aufgabenumfang reibungslos arbeitet.

tens et al., 2005). Der aus der Lizenzgestaltung erwachsende Vorteil offener gegenüber proprietärer Software bezieht sich also nur auf einen Teil aller entscheidungsrelevanten Kosten, die nachfolgend wie üblich als "Total Cost of Ownership" (TCO) bezeichnet werden.

Nachfolgend werden diese Kriterien genauer diskutiert und ihre empirische Relevanz für die Nutzungsentscheidung beschrieben.

2.5.2.1 Kostenmotiv

Die Frage, ob der Lizenzkostenvorteil offener Software auch bei einer Betrachtung der TCO bestehen bleibt, ist pauschal nicht zu beantworten. Zwar existieren diverse vergleichende Studien,[79] diese kommen aber zu uneinheitlichen Ergebnissen, da sie jeweils solche Nutzungsszenarien zugrundelegen, die den jeweiligen Auftraggebern genehme Ergebnisse sicherstellen. Kurz: Microsoft kann nachweisen, daß für bestimmte Szenarien der Einsatz von Windows weniger Kosten verursacht als der von Linux, während IBM durch die Wahl anderer Szenarien dasselbe für Linux gelingt.

Eine *unabhängige* vergleichende Kostenstudie hat die Unternehmensberatung Unilog (2003) im Auftrag der Stadt München durchgeführt.[80] Anlaß war die Erfordernis, die rund 15.000 Arbeitsplätze der Stadtverwaltung von älteren Versionen der Microsoft-Produkte Windows und Office entweder auf neuere Versionen derselben Software oder aber auf eine Kombination von Linux und OpenOffice.org umzustellen. Die wesentlichen Ergebnisse waren:

- Die TCO der Microsoft-Lösung über den Planungszeitraum von fünf Jahren betragen 34,2 Mio. €, die der Linux-OpenOffice-Lösung 45,8 Mio. €.

[79] Eine Übersicht über verschiedene vergleichende Kostenstudien geben Brügge et al. (2004).

[80] Die Studie wurde gemäß vom Bundesministerium des Innern (2001) vorgegebener Empfehlungen für die Durchführung von Wirtschaftlichkeitsbetrachtungen beim Einsatz von IT durchgeführt.

- Die Kosten für die Migration bestehender Software, die Umstellung von Formularen und Makros, Schulungen sowie die Einarbeitung betragen bei der Microsoft-Lösung 20,8 Mio. €, bei der Linux-OpenOffice-Lösung 40,8 Mio. €. Daraus folgt, daß die durch einen Wechsel von Microsoft zu Linux und OpenOffice anfallenden *Wechselkosten* rund 20 Mio. € betragen.

- Die TCO der Microsoft-Lösung enthalten an Microsoft zu zahlende Lizenzgebühren in Höhe von 7,3 Mio. €.

Offensichtlich wurden in diesem Fall die Lizenzkostenvorteile offener Software durch die vergangenheitsbedingten Wechselkostennachteile zunichte gemacht. Ohne diesen Nachteil wäre die auf offener Software basierende Lösung allerdings eben wegen des Fehlens von Lizenzgebühren kostengünstiger gewesen.[81]

2.5.2.2 Strategische und operative Motive

Unter strategischen Gesichtspunkten weist offene Software gegenüber proprietärer inhärente Vorteile auf. Zwar kann auch ihr Einsatz zu hohen Wechselkosten und Lock-In führen, in Ermangelung von Exklusivrechten ist es aber niemandem möglich, diese Situation monetär auszubeuten. Des weiteren erhöht die Verfügbarkeit des Quelltextes die Möglichkeiten zur Einflußnahme auf die Weiterentwicklung der Software sowie die Zukunftssicherheit.

Insbesondere der letztgenannte Aspekt ist in Anbetracht der Dynamik von Softwaremärkten hervorzuheben: Neue Produkte und auch Hersteller verbleiben oft nur sehr kurz im Markt. So untersuchen Fosfuri und Giarratana (2004) die Entwicklung von Eintritten in und Austritten aus dem noch jungen Markt für Sicherheitssoftware. Ihre in Abbildung 2.5 dargestellten Ergebnisse legen nahe, daß ein Großteil der in der 90er Jahren angebotenen Sicherheitssoftware inzwischen aufgrund des Austritts von Herstellern verwaist ist. Offensichtlich ist

[81] Da die Linux-OpenOffice-Lösung aber hinsichtlich strategischer und operativer Kriterien Vorteile aufwies, entschied die Stadt München, die hohen Wechselkosten inkaufzunehmen und auf offene Software umzustellen.

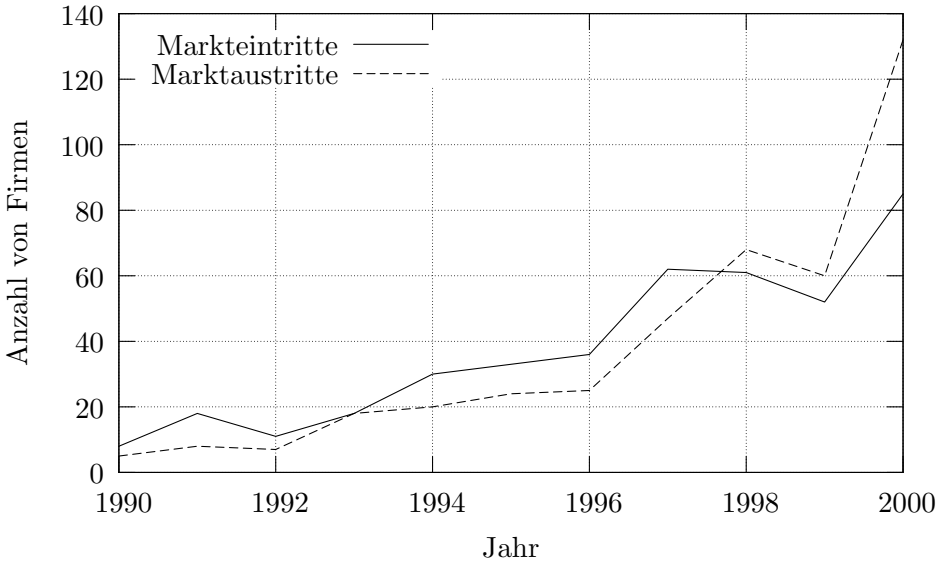


Abbildung 2.5: Entwicklung von Ein- und Austritten in bzw. aus dem Markt für Sicherheitssoftware (Fosfuri und Giarratana, 2004, S. 31)

offene Software vor einer derartigen Verwaisung in gewisser Hinsicht geschützt, da sie bei Interesse von jedermann weiterentwickelt werden kann.

Ob offene Software gegenüber proprietärer auch operative Vorzüge aufweist, ist hingegen pauschal nicht zu beantworten. Zahlreiche Leistungsvergleiche[82] legen nahe, daß hier keine systematischen Vor- und Nachteile existieren und eine fallweise Betrachtung unerläßlich ist. Im Rahmen dieser Arbeit wird daher angenommen, beide Arten von Software seien operativ gleichwertig.

2.5.2.3 Empirische Bedeutung der Motive

Eine Untersuchung zur empirischen Bedeutung der oben diskutierten Motive für die Nutzungsentscheidung von Unternehmen hat Mortali (2003) vorgelegt. Er hat 62 für den Einsatz von IT verantwortliche Vorstände zu den wahrgenommenen Vor- und Nachteilen offener Software gegenüber proprietärer befragt; Tabelle 2.11 gibt die jeweils wichtigsten wieder. Die Prozentangaben beziehen sich

[82] Eine umfangreiche Zusammenstellung von Leistungsvergleichen bezüglich verschiedener operativer Kriterien findet sich in Wheeler (2005).

Vorteile		Nachteile	
Geringere TCO	46 %	Schlechter Support	40 %
Geringere Lizenzkosten	39 %	Geringer Erprobungsgrad	19 %
Kein Lock-In	39 %	Keine einschlägigen Fähigkeiten	18 %
Verfügbarkeit des Quelltextes	25 %	Keine etablierten Beurteilungskriterien	17 %
Möglichkeit zur Weiterentwicklung	17 %	Probleme mit der Integration und Kompatibilität	15 %
Nutzung offener Standards	16 %	Umstellungskosten	14 %

Tabelle 2.11: Wahrgenommene Vor- und Nachteile offener Software

dabei auf den Anteil der Befragten, welche das jeweilige Motiv als inhärenten Vor- bzw. Nachteil ansehen, und zwar unabhängig davon, ob ihr Unternehmen aufgrund der spezifischen Situation tatsächlich offene oder proprietäre Software einsetzt. Die wichtigsten Vorteile offener Software liegen demnach im monetären[83] und strategischen Bereich, die wichtigsten Nachteile sind schlechter Support, mangelnde Erfahrung und Wechselkosten.

Goode (2005) kommt in einer Umfrage unter 500 australischen Unternehmen zu ähnlichen Ergebnissen. Demnach ist der Hauptgrund, offene Software nicht einzusetzen, die geringe Relevanz für das unternehmensspezifische Anwendungsszenario, gefolgt vom Mangel an Support. Und auch Brügge et al. (2004) können auf der Grundlage fünf weiterer, qualitativ ausgewerter Studien diese Ergebnisse weitestgehend bestätigen.

2.5.3 Markterfolg

Ein weiteres gutes Maß für die Bedeutung offener Software aus Nutzersicht ist zweifelsfrei ihr Markterfolg, der sich, da offene Software selbst nicht mit Marktpreisen bewertet werden kann, nicht anhand von Marktanteilen, sondern nur über Nutzungsraten diskutieren läßt. Allerdings sind auch solche Nutzungsra-

[83] Insgesamt bezeichnen 68 % der Befragten die Reduktion der TCO als ihre hauptsächliche Aufgabe. Der wahrgenommene Kostenvorteil offener Software ist also für die letztliche Nutzungsentscheidung von herausragender Bedeutung.

ten nur schwer zu ermitteln; die Verwendung offener Software hinterläßt naturgemäß weniger "Spuren" als die proprietärer.

Grundsätzlich gelten Unternehmen und öffentliche Einrichtungen als intensivere Nutzer als Privatpersonen (vgl. Feller und Fitzgerald, 2002). Eine Ursache dafür ist wohl, daß große Organisationen üblicherweise ein und dieselbe Softwareausstattung auf sehr vielen Rechnern installieren, so daß der Lizenzkostenvorteil offener gegenüber proprietärer Software besonders zum Tragen kommt, wohingegen einmalige Kosten an Bedeutung einbüßen.

Verschiedene Studien haben sich deshalb ganz gezielt der Nutzung offener Software durch Unternehmen und öffentliche Einrichtungen angenommen: Wichmann (2002) befragte insgesamt 1.452 derartige Organisationen in Deutschland, Großbritannien sowie in Schweden und ermittelte Nutzungsraten von 44 %, 32 % bzw. 18 %. Des weiteren stellt er fest, daß 54 % aller Organisationen, die offene Software verwenden, deren Bedeutung für die eigene IT-Landschaft mindestens als durchschnittlich einstufen. Für Großbritannien stützt Mortali (2002) diese Ergebnisse mit einer Umfrage unter 76 mit IT betrauten Vorstandsmitgliedern, nach denen 37 % der Unternehmen bereits offene Software einsetzen und weitere 12 % ihren Einsatz planen. Dabei liegt das Haupteinsatzfeld im Bereich der Infrastruktur; direkt an den Arbeitsplätzen verwenden nur wenige Unternehmen offene Software. D'Antoni (2004) beleuchtet den Einsatz offener Software in den USA. Hier verwenden laut einer Umfrage unter 420 Unternehmen 60 % eine Mischung aus offener und proprietärer Software, wohingegen 38 % primär proprietäre und 2 % primär offene Software einsetzen. Des weiteren ergibt sich, daß große Unternehmen intensivere Nutzer offener Software sind als kleine.

Aus Nutzersicht spielt offene Software also bereits heute eine erhebliche Rolle, die sich zudem wohl noch ausweiten wird.[84] Diverse von Wheeler (2005) zusammengestellte Studien, die nicht auf das Gesamtphänomen, sondern auf den

[84] Laut Broersma (2004) erwartet das Marktforschungsinstitut IDC bis zum Jahr 2008 eine Verdopplung der Nutzungsraten offener Software.

Markterfolg offener Software in verschiedenen Teilmärkten abzielen, bestätigen diese Sicht.

2.5.4 Unternehmensspezifische Nutzungformen

Des weiteren ergeben sich aus der Verfügbarkeit der Quelltexte für Unternehmen zwei Nutzungsformen offener Software, die für proprietäre Software nicht denkbar sind: die kostenfreie Verwertung offener Software bei der Produktentwicklung sowie ihre Anpassung an unternehmensspezifische Prozesse.

Laut Brügge et al. (2004) vollzieht sich die *Verwertung bei der Produktentwicklung* grundsätzlich in drei Phasen: der Identifikationsphase, in der eine geeignete offene Software ermittelt wird, der Adaptionsphase, in der die offene Software an die Eigenentwicklung angebunden wird, und der Modifikationsphase, in der die offene Software an die eigenen Bedürfnisse angepaßt wird. Zudem versuchen Unternehmen nach der Modifikationsphase regelmäßig, die Führung des Entwicklungsprojekts für die Aufnahme der Veränderungen in den offiziellen Quelltext zu gewinnen, um das zukünftige Zusammenspiel von offener Software und Eigenentwicklung reibungsloser zu gestalten. Das unternehmerische Motiv für die eben vorgestellte Nutzung offener Software ist offensichtlich die Reduktion von Kosten. Ein Beispiel gibt das Unternehmen Apple mit seinem Betriebssystem "Mac OS X", welches in Teilen auf BSD Unix basiert.[85]

Seltener vorzufinden ist hingegen die *Anpassung an unternehmensspezifische Prozesse*. Dabei wird eine geeignete offene Software auf der Quelltextebene an spezifische, von kommerziellen Produkten nicht befriedigte Informationsverarbeitungsbedürfnisse angepaßt, um bestmögliche Effizienz zu erzielen. Man beachte, daß für diese Nutzungsform die von Copyleft-Lizenzen auferlegte Pflicht zur Offenlegung der Quelltexte irrelevant ist, da sie erst im Fall der Weiterverbreitung der Software wirksam wird. Ein Beispiel stellt die Suchmaschine Google dar. Das unternehmerische Rückgrat dieses Unternehmens bildet of-

[85] Mehr Informationen zur Integration offener Software in Mac OS X finden sich unter `www.opendarwin.org`.

fensichtlich die Fähigkeit, eine gewaltige Menge von Daten hoher Spezifizität effizient verwalten und verteilen zu können;[86] schlüsselfertige Produkte, die derartiges leisten, gibt es nicht zu kaufen. Daher verwendet Google laut Wagner (2000, 2001) selbstentwickelte Software, der das bereits vorgestellte Linux zugrundeliegt.

2.6 Zwischenfazit

Obgleich offene Software insgesamt ein äußerst vielfältiges Phänomen darstellt, stechen dennoch drei Aspekte deutlich hervor: Erstens ist die besondere Lizenzgestaltung zu nennen, die den Nutzern deutlich weitreichendere Rechte als üblich einräumt und Lizenzgebühren verbietet. Bemerkenswert ist zweitens aber auch der schnelle, iterative Prozeß, in dem offene Software von meritokratisch aufgebauten Gemeinschaften Freiwilliger weiterentwickelt wird. Und drittens ist schließlich auch die historisch gewachsene Hacker-Kultur hervorzuheben, welche die Entwicklung offener Software mit ihren speziellen Werten und Normen ganz wesentlich trägt. Wer das in dieser Arbeit behandelte Phänomen umfassend verstehen will, darf sicherlich keinen dieser Aspekte vernachlässigen.

Des weiteren erweist sich das Bild von der Spielzeugsoftware, die von langhaarigen Idealisten und pickligen Teenagern zusammengebastelt wird, als Mythos. Offene Software wird professionell entwickelt, hat eine hohe Qualität und wird dementsprechend vielfältig eingesetzt.

[86] Laut Nielsen//NetRatings (2004) erreicht Google 47 % der europäischen Internetnutzer, welche, da sie überdurchschnittlich aktiv sind, sogar 69 % aller Suchanfragen stellen.

3 Entwicklung offener Software

In diesem Kapitel wird die Entwicklung offener Software aus ökonomischer Sicht analysiert. Dazu werden zunächst verschiedene, vornehmlich empirische Arbeiten herangezogen, die erklären, welche Motive Entwickler zu einer Beteiligung an offenen Softwareprojekten bewegen. Anschließend wird mithilfe ausgewählter Modelle beleuchtet, wie diese Motive in *bereits bestehenden* offenen Entwicklungsgemeinschaften wirken und wie die innerhalb einer Gemeinschaft auftretenden Koordinationsprobleme gelöst werden. Den Schwerpunkt dieses Kapitels bildet aber Abschnitt 3.3. Er beinhaltet ein auf Gutsche (2005b) aufbauendes Modell dazu, wie und unter welchen Voraussetzungen *neue* Entwicklungsgemeinschaften entstehen können. Daraus wird abschließend abgeleitet, welche Arten von Software ökonomisch nachhaltig gemäß der offenen Basar-Methode entwickelt werden können und welche eine Domäne proprietärer Softwarehersteller bleiben werden.

3.1 Motive der Entwickler

3.1.1 Einzelpersonen

Einzelpersonen tragen, wie in Abschnitt 2.3 gesehen, in etwa die Hälfte der Entwicklungsanstrengungen. Ihre Motive werden im folgenden kategorisiert und ihre empirische Relevanz beschrieben.

In der psychologischen Literatur wird grundsätzlich zwischen zwei Gruppen von Motiven für menschliches Handeln unterschieden: intrinsischen und extrinsischen Motive (vgl. etwa Ryan und Deci, 2000).

Intrinsische Motive wirken dann, wenn eine Tätigkeit inhärent befriedigend ist und folglich auch ohne einen externen Anreiz gerne ausgeführt wird. Insbesondere in der Presse wird die Entwicklung offener Software häufig allein auf intrinsische Motive zurückgeführt; genannt werden dabei am häufigsten Altruismus, Spaß und ein gewisses politisches, antikapitalistisches Sendungsbewußtsein. Nachfolgend wird aber gezeigt, daß die alleinige Berücksichtigung intrinsischer Motive in die Irre führt.

Extrinsische Motive fußen hingegen auf einem von außen gesetzten Anreiz. Ihnen wurde im Zusammenhang mit der Entwicklung offener Software zunächst jede Bedeutung abgesprochen, da das einfachste extrinsische Motiv, die monetäre Entlohnung, zu fehlen scheint. In Abschnitt 2.3 wurde aber bereits erläutert, daß ein erheblicher Anteil der Entwickler offener Software für ihre Arbeit bezahlt wird, so daß sich die Frage nach den Motiven zum Teil auf die beauftragenden Unternehmen verschiebt. Zudem treten bei genauerem Hinsehen zahlreiche weitere extrinsische Motive zutage: Lerner und Tirole (2002) etwa betonen die Bedeutung von Karrieremotiven und nennen diesbezüglich den Aufbau von Humankapital sowie Job-Market-Signaling. Raymond (2001) gibt als Praktiker zwei weitere Motive an, die sich unmittelbar auf die Entwicklung und Nutzung einer offenen Software beziehen: die Befriedigung eines vom Markt nicht adressierten Eigenbedarfs und die Reduktion von Entwicklungskosten durch die Nutzung der als besonders effizient erachteten Basar-Methode. In der Literatur werden noch weitere denkbare Motive genannt, die hier aufzuzählen allerdings müßig ist; eine gute Zusammenfassung geben Feller und Fitzgerald (2002).

Aufbauend auf diesen Vorüberlegungen haben diverse Studien versucht, mittels Umfragen die tatsächlichen Motive von Entwicklern zu ermitteln. Die wichtigsten Ergebnisse werden nachfolgend zusammengefaßt. Dabei wurden alle zitierten Studien bereits in Abschnitt 2.3 herangezogen; für Details über den Umfang und den befragten Personenkreis sei daher auf Tabelle 2.4 verwiesen.

Hars und Ou (2002) untersuchen die Bedeutung von insgesamt acht Motiven: dem Wunsch nach Selbstbestimmung, dem Wunsch nach Gemeinschaft und Altruismus (alle intrinsisch) sowie der Bildung von Humankapital, dem Wunsch

	Alle		Bezahlt	
Motiv	relative Häufigkeit	Korrelation mit Zeiteinsatz	relative Häufigkeit	Korrelation mit Zeiteinsatz
Intrinsisch				
Selbstbestimmung	80 %	0,07	62 %	0,22
Gemeinschaftsgefühl	28 %	0,12	31 %	-0,31
Altruismus	17 %	0,19	8 %	-0,16
Extrinsisch				
Humankapital	88 %	0,14	85 %	0,07
Externe Anerkennung	43 %	-0,02	46 %	-0,18
Eigenbedarf	39 %	0,30	39 %	0,33
Selbstvermarktung	37 %	0,32	69 %	0,42
Verkauf von Produkten	14 %	0,36	54 %	0,30

Tabelle 3.1: Motive von Entwicklern offener Software (Hars und Ou, 2002)

nach externer Anerkennung, der Befriedigung von Eigenbedarf, Selbstvermarktung und dem Verkauf von Produkten. Die Motive wurden in der Weise operationalisiert, daß die Befragten den Grad ihrer Zustimmung zu verschiedenen, den Motiven zugeordneten Aussagen angeben mußten, und zwar auf einer Skala von 1 (starke Ablehnung) bis 7 (starke Zustimmung). Tabelle 3.1 faßt die Ergebnisse zusammen. Dabei gibt die relative Häufigkeit den Anteil der Befragten mit einer Zustimmung von 6 oder 7 an und die Korrelation mit dem Zeiteinsatz den Zusammenhang zwischen dem Grad der Zustimmung und der durchschnittlichen wöchentlichen Arbeitszeit. Zudem sind alle Ergebnisse auch für die Teilmenge derjenigen ausgewiesen, die für ihre Arbeit an offener Software bezahlt werden.

Offensichtlich wirken extrinsische Motive grundsätzlich deutlich stärker als intrinsische, und zwar unabhängig davon, ob man ihre Bedeutung an der relativen Häufigkeit oder an der Korrelation mit dem Zeiteinsatz bemißt. Des weiteren sind, wie zu erwarten war, bezahlte Entwickler stärker extrinsisch motiviert als Freiwillige. Aber auch das intrinsische Motiv der Selbstverwirklichung spielt durchgängig eine Rolle.

Motiv	Alle	Klasse 1	Klasse 2	Klasse 3	Klasse 4
Intellektuelle Stimulation	45 %	41 %	45 %	69 %	12 %
Humankapital	41 %	20 %	43 %	72 %	19 %
Bezahlung	40 %	86 %	18 %	26 %	32 %
Beruflicher Eigenbedarf	34 %	91 %	8 %	12 %	28 %
Glaube an offene Software	33 %	12 %	22 %	42 %	64 %
Privater Eigenbedarf	30 %	11 %	100 %	0 %	2 %
Verpflichtung aus Nutzung	29 %	23 %	20 %	6 %	83 %
Teamarbeit	20 %	17 %	16 %	28 %	19 %
Beruflicher Status	18 %	25 %	6 %	22 %	18 %
Externe Anerkennung	11 %	14 %	8 %	11 %	13 %
Abneigung gegen proprietäre Software	11 %	11 %	8 %	9 %	19 %
Anteil an Grundgesamtheit	100 %	25 %	27 %	29 %	19 %

Tabelle 3.2: Motive von Entwicklern offener Software (Lakhani und Wolf, 2005)

Gosh et al. (2002) kommen zu vergleichbaren Ergebnissen. Des weiteren untersuchen sie, inwieweit sich die Motive für den Eintritt in eine offene Entwicklergemeinschaft von denen für den Verbleib in derselben unterscheiden. Sie finden keine signifikanten Abweichungen, woraus folgt, daß die Entwickler alle relevanten Anreize bereits ex ante gut zu erfassen vermögen.

Die aggregierte Betrachtungsweise der beiden oben angeführten Studien verleitet möglicherweise zu der Ansicht, alle Entwickler seien grundsätzlich ähnlich motiviert. Daß diese Ansicht falsch ist, zeigen Lakhani und Wolf (2005) mit einer differenzierteren Betrachtung. Als Grundlage dient ihnen eine Befragung, bei der Entwickler verschiedener Projekte aus mehreren vorgegebenen Motiven die drei mit der größten Bedeutung auswählen sollten. Anschließend wurden die Befragten anhand ihrer Antworten mittels einer Clusteranalyse[1] in vier Motivklassen eingeteilt. Die Ergebnisse sind in Tabelle 3.2 angegeben.

Eine griffige Charakterisierung der Klassen ergibt sich, wenn man jeweils diejenigen Motive betrachtet, die von mehr als der Hälfte aller Klassenangehörigen

[1]Mittels einer Clusteranalyse wird eine Grundgesamtheit so in Klassen eingeteilt, daß jede Klasse für sich genommen ähnliche Objekte enthält, die Klassen zueinander aber möglichst unähnlich sind. Eine genaue Beschreibung dieser Methode geben etwa Backhaus et al. (2005).

genannt wurden: Demnach sind die in Klasse 1 enthaltenen Entwickler insbesondere durch beruflichen Eigenbedarf und die Bezahlung ihrer Mitarbeit motiviert; man kann sie daher als Professionals bezeichnen. Für Klasse 2 liegt der Fokus hingegen auf privatem Eigenbedarf; sie umfaßt also die Hobbyisten. Die Entwickler aus Klasse 3 nennen mehrheitlich den Aufbau von Humankapital und die intellektuelle Stimulation; sie stellen sich somit als Lernende dar. Die Entwickler aus Klasse 4 schließlich fühlen sich durch die Nutzung offener Software zur Mitarbeit verpflichtet und glauben an die moralische Überlegenheit offener Software; hier sind offensichtlich die Gläubigen versammelt, welche auch den Kern der FSF bilden.

Die Motive der Entwickler sind erkennbar so divers, daß jede Analyse, die sich auf eine Motivgruppe beschränkt, Stückwerk bleiben muß. Ähnlich sind sich nahezu alle Motive aber darin, daß sie erst dann zu wirken beginnen, wenn die Entwicklungsgemeinschaft eine gewisse Grösse erreicht hat: Ein Unternehmen wird nur dann bereit sein, für die Arbeit an offener Software zu bezahlen, wenn jene durch die Mitarbeit vieler markttauglich gemacht wurde. Eine für den Eigenbedarf entwickelte spezielle Funktion wird erst dann nützlich, wenn sie in ein größeres Softwaresystem integriert werden kann, sonst hängt sie in der Luft. Der Aufbau von Humankapital gelingt letztlich nur durch die Arbeit in einem industriellen Maßstab und die regelmäßige Begutachtung der eigenen Arbeit durch andere. Und auch Job-Market-Signaling erfordert etablierte Institutionen. Allein die Gläubigen bedürfen durch ihre vergleichsweise starke intrinsische Motivation nicht in dem Maße einer großen Entwicklungsgemeinschaft. Aber auch ihr Hauptmotiv, das Gefühl der aus der Nutzung offener Software erwachsenden moralischen Verpflichtung, setzt die Existenz einer nutzbaren Software voraus.[2]

Insgesamt wird deutlich, daß die Entwicklung offener Software aus ökonomischer Sicht im Kern keine Frage der Anreize sondern der Koordination beinhaltet. Warum entwickeln Tausende von fähigen Programmierern offene Software? Weil

[2]Man kann dieses Motiv auch als Kooperation in einem wiederholten Spiel betrachten.

es ihren Präferenzen entspricht! Wann aber gelingt Tausenden von Programmierern die Koordination auf die Entwicklung offener Software zum allseitigen Nutzen? In Abschnitt 3.3 wird ein Modell entwickelt, welches genau diese Frage beleuchtet.

3.1.2 Motive von Unternehmen

Wie in Kapitel 2 gesehen interessiert sich ein Unternehmen aus einem von drei Gründen für die Entwicklung offener Software: erstens, da es komplementäre Produkte anbietet, zweitens, da es offene Software in eigene Produkte integriert, oder drittens, da es die Software in unternehmensinternen Prozessen einsetzt. Wiederum setzen alle drei Gründe voraus, daß die offene Software von einer ausreichend großen Entwicklergemeinschaft getragen wird und kommerziell einsetzbar ist. Der für den Fortgang dieser Arbeit wichtigste Aspekt der Motive von Unternehmen ist somit bereits etabliert; die nachfolgende Darstellung wird dementsprechend kurz gehalten.

Die Entscheidung für oder gegen die Beteiligung an einem offenen Entwicklungsprojekt fällen Unternehmen selbstverständlich auf der Grundlage einer nüchternen Abwägung von Kosten und Nutzen (vgl. Hawkins, 2004). Die dafür wichtigsten Kosten- und Nutzenkomponenten sind in Anlehnung an Feller und Fitzgerald (2002), Brügge et al. (2004) und Henkel (2004a) in Tabelle 3.3 zusammengefaßt. Insgesamt stehen dem Hauptnachteil, daß mit den eigenen Beiträgen selbst keine Lizenzumsätze erzielt werden können, offenbar viele Vorteile gegenüber. Welche Seite überwiegt, ist pauschal nicht zu sagen.[3]

Die Literatur zur empirischen Relevanz der in Tabelle 3.3 aufgeführten Motive ist derzeit noch schmal. Laut einer von Bonaccorsi und Rossi (2005) durchgeführten Befragung von 146 italienischen Unternehmen ist das Hauptmotiv die aus der Einbindung externer Entwickler resultierende Senkung der Kosten für Forschung und Entwicklung, gefolgt von dem Wunsch, hochwertige offe-

[3] Behlendorf (1999) beschreibt eine beispielhafte Kosten-Nutzen-Abwägung mit realistischen Zahlen, welche zugunsten offener Software ausfällt.

Grund für Beteiligung	Nutzenkomponenten	Kostenkomponenten
Alle	• externe Entwicklungsunterstützung und Kostenreduktion • Standardsetzung • Erhöhte Programmierdisziplin • Reputationsgewinn bei gutem Quelltext • Guter Ruf in OSS-Szene • Besserer Zugriff auf potentielle neue Mitarbeiter	• Verlust von Wettbewerbsvorteilen • Verlust der Kontrolle über geistiges Eigentum • Reputationsverlust bei schlechtem Quelltext • Verletzungen von Schutzrechten liegen offen • Kosten für Kommunikation mit externer Entwicklergemeinschaft
Verkauf von Komplementen	• Erhöhte Nachfrage nach Komplement	• Verlust von Möglichkeiten zum Bundling
Integration in eigene Produkte	• Nutzung von gemäß Copyleft lizensiertem Quelltext möglich • Schaffung von Kundennutzen durch Offenlegung des Quelltexts	
Verwendung in internen Prozessen	• Geringere Abhängigkeit von externen Entwicklern	• Einsicht für Wettbewerber in Geschäftsprozesse • Sichtbarwerden von Sicherheitslücken

Tabelle 3.3: Potentielle Nutzen- und Kostenkomponenten bei der Beteiligung eines Unternehmens an der Entwicklung offener Software

ne Software lizenzgebührenfrei in eigene Produkte integrieren zu können. Viele Unternehmen behaupten des weiteren, sie würden sich auch aufgrund ihrer Übereinstimmung mit den Werten der Hacker-Kultur beteiligen. Eine flankierende Untersuchung des tatsächlichen Verhaltens der befragten Unternehmen entlarvt diese Aussage aber oft als reines Lippenbekenntnis.

Während die Studie von Bonaccorsi und Rossi (2005) Unternehmen jeder Coleur einbezieht, hat Henkel (2004c) eine enger abgegrenzte, aber auch präzisere Untersuchung vorgelegt. Er beschränkt sich auf Hersteller von Hardware für Embedded-Linux; die 137 befragten Unternehmen beteiligen sich also vornehmlich in ihrer Eigenschaft als Anbieter eines Komplements. Als wichtigste Motive für ihre Beteiligung an der Entwicklung von Embedded-Linux nennen sie den

von der GPL ausgeübten Zwang,[4] die Pflege des eigenen Rufs sowie die Senkung der Kosten durch die externe Unterstützung bei Fehlerbereinigung, Weiterentwicklung und Wartung. Am unwichtigsten sind dagegen die Identifikation neuer Mitarbeiter sowie die Setzung vorteilhafter Standards.

3.2 Modelle zur Entwicklung offener Software

3.2.1 Die Rolle der Befriedigung von Eigenbedarf

In den ersten formalen Modellen wurde die Entwicklung offener Software als Form der privaten Bereitstellung öffentlicher Güter aufgefaßt. Dabei fungieren alle Spieler sowohl als Nutzer als auch als potentielle Entwickler einer offenen Software. In ihrer Rolle als Entwickler können sie die Software verbessern, haben aber die dafür anfallenden Kosten selbst zu tragen. In ihrer Rolle als Nutzer profitieren sie aufgrund eines Eigenbedarfs sowohl von den eigenen Verbesserungen als auch von denen der anderen Spieler.

Während die Interpretation von offener Software als öffentlichem Gut aus Nutzersicht den Kern trifft, greift die implizierte alleinige Berücksichtigung des Motivs der Befriedigung von Eigenbedarf auf der Entwicklerseite wie gesehen zu kurz. Dennoch werden nachfolgend zwei wichtige Modelle dieser Art kurz vorgestellt.

3.2.1.1 Das Modell von Johnson (2002)

Johnson (2002) betrachtet ein statisches Spiel zwischen $n \in \mathbb{N}$ Spielern um die Frage, wer eine allen bekannte mögliche Erweiterung einer bestehenden offenen Software entwickelt. Dabei hat jeder Spieler die Wahl zwischen den beiden Aktionen, die Erweiterung selbst zu entwickeln oder nichts zu tun. In seiner Rolle als Nutzer erhält jeder Spieler $i \in \{1, \ldots, n\}$ die Auszahlung v_i, falls die Erweiterung von mindestens einem Spieler entwickelt wird. Des weiteren muß

[4] Bei Software, die unter keiner Copyleft-Lizenz steht, entfällt dieser Zwang.

jeder Spieler in seiner Rolle als Entwickler die Kosten c_i tragen, wenn er sich selbst zur Entwicklung der Erweiterung entschließt.

Die jeden Spieler charakterisierenden Tupel (c_i, v_i) stellen private Informationen dar; sie sind unabhängige Realisationen der gemäß der Verteilungsfunktion G verteilten Zufallsvariablen (c, v). Dabei habe G die Trägermenge $[c_L, c_H] \times [v_L, v_H]$ mit $c_L > 0$ und $v_H \geq 0$; des weiteren sei G stetig und auf der gesamten Trägermenge streng monoton steigend. Zudem bezeichne F die von G induzierte Verteilungsfunktion des Nutzen-Kosten-Verhältnisses v/c, d. h. $F(q) = \Pr\{v/c < q\}$. Es gelte $F(1) < 1$, da sonst wegen $\Pr\{v < c\} = 1$ niemals entwickelt würde.

Das geeignete Lösungskonzept für dieses Spiel ist offensichtlich das des Bayesianischen Nash-Gleichgewichts. Zudem beschränkt sich Johnson in seiner Analyse der Einfachheit halber auf symmetrische Gleichgewichte.[5] Er zeigt, daß ein eindeutiges Gleichgewicht existiert, welches durch drei Wahrscheinlichkeiten charakterisiert werden kann: die Wahrscheinlichkeit p_n^*, mit der ein Spieler i entwickelt, die Wahrscheinlichkeit π_n^*, mit der mindestens ein Spieler $j \neq i$ entwickelt,[6] sowie die Wahrscheinlichkeit γ_n^*, daß kein Spieler entwickelt. Für die drei Wahrscheinlichkeiten gilt definitionsgemäß $\gamma_n^* = (1-p_n^*)^n = (1-p_n^*)(1-\pi_n^*)$.

Johnson interessiert sich nun insbesondere für den Zusammenhang zwischen der Anzahl der Spieler und den oben definierten Wahrscheinlichkeiten. Dieses Interesse resultiert aus der Überlegung, daß eine Erhöhung der Spieleranzahl zum einen einen positiven Effekt auf die Entwicklung haben könnte, da schlicht mehr potentielle Entwickler vorhanden sind, zum anderen aber auch einen negativen, da sich der Anreiz zum Trittbrettfahren erhöht. Nachfolgend werden einige Resultate vorgestellt, die das Zusammenspiel dieser beiden gegenläufigen Effekte erhellen:[7]

[5] Alle Spieler verwenden also dieselbe auf ihre privaten Informationen konditionierte Strategie.

[6] Die Wahrscheinlichkeiten p_n^* und π_n^* sind für alle Spieler wegen der Beschränkung auf symmetrische Gleichgewichte gleich. Sie sind daher nicht indiziert.

[7] Johnson analysiert noch einige Modellvarianten, die aber aus Platzgründen hier nicht vorgestellt werden.

- Die für jeden Spieler i identische Wahrscheinlichkeit π_n^* dafür, daß einer der anderen entwickelt, steigt in n. Da sich folglich der Nutzen des Nichtstuns erhöht, steigt auch der Erwartungsnutzen jedes Spielers.

- Die Wahrscheinlichkeit p_n^*, mit der ein Spieler selbst entwickelt, fällt hingegen. Dies ist Ausdruck klassischen Trittbrettfahrerverhaltens.

- Die Entwicklungswahrscheinlichkeit $1 - \gamma_n^*$ kann abhängig von F sowohl zu- als auch abnehmen. Mit $-1/((n-1)e)$ existiert aber eine untere Schranke für ihre Abnahme $(1 - \gamma_{n+1}^*) - (1 - \gamma_n^*)$. Zudem konvergiert diese Schranke für $n \to \infty$ gegen 0.

- Des weiteren konvergiert die Entwicklungswahrscheinlichkeit $1 - \gamma_n^*$ für $n \to \infty$ gegen $1 - c_L/v_H > 0$. Auch bei sehr vielen Spielern bricht die Weiterentwicklung also nicht vollständig zusammen.

- Die erwartete Anzahl der Spieler, welche die Erweiterung entwickeln, konvergiert für $n \to \infty$ gegen $\ln(v_H/c_L)$. Trittbrettfahrten haben also den unerwarteten Nebeneffekt, die Zahl unnötiger Mehrfachentwicklungen und somit auch die Gesamtkosten nach oben zu beschränken.

Die Wirkung des Trittbrettfahrereffekts in einem Projekt mit vielen, nur durch einen Eigenbedarf motivierten Entwicklern beschreibt das vorliegende Modell zweifelsfrei sehr gut. Es weist aber im Licht von Kapitel 2 mehrere gravierende Schwächen auf:

Erstens berücksichtigt es nur eines von vielen Motiven. Andere Motive, die ja oft nicht auf den Nutzwert des Endergebnisses sondern auf den des Selbermachens abzielen, könnten die Annahme $c_L < 0$ rechtfertigen; die oben beschriebenen Ergebnisse sind aber gegenüber einer solchen Änderung der Annahmen nicht robust. Zweitens erscheint die Beschränkung auf symmetrische Gleichgewichte und identisch verteilte Nutzen und Kosten unangemessen. Dazu stelle man sich etwa vor, das Modell beschreibe die Weiterentwicklung von OpenOffice.org und Spieler 1 sei das Unternehmen Sun, der Initiator dieses Projekts. Unter diesen Voraussetzungen gilt wohl für viele denkbare Programmerweiterungen

$\Pr\{v_1 > c_1\} = 1$; es ist dann ein naheliegendes asymmetrisches Gleichgewicht, daß Sun immer entwickelt, und die anderen nie. Das vom vorliegenden Modell erfaßte Koordinationsproblem ist also in der Realität oft gar nicht vorhanden. Drittens schließlich bietet das Modell keine Erklärung für die eigentliche Frage, wann und warum die Entwickler offener Software geistiges Eigentum verschenken. Den Spielern im Modell bleibt aufgrund der vorgegebenen Aktionen keine andere Wahl. Mag diese Einschränkung im Fall von Copyleft-Lizenzen akzeptabel sein, so ist sie für Non-Copyleft-Lizenzen völlig unangebracht. Die Entwickler verfügen dann nämlich über die Möglichkeit, ihre Eigenentwicklung geheimzuhalten und an die anderen Spieler zu verkaufen. Machte man den Spielern im vorliegenden Modell eine entsprechende Aktion verfügbar, so würde diese die des Verschenkens schwach dominieren, und zwar unabhängig von der Modellierung des Markts für die Erweiterung. Insgesamt scheint das Modell von Johnson daher am Kern der Sache vorbeizugehen.

3.2.1.2 Das Modell von Myatt und Wallace (2002)

Myatt und Wallace (2002) betrachten insgesamt vier eng verwandte statische Spiele mit zwei Spielern[8] und vollständigen Informationen. Wiederum fungieren beide Spieler sowohl als Nutzer als auch als Entwickler und haben die Wahl zwischen den beiden Möglichkeiten, nicht beizutragen (Aktion 0) oder aber beizutragen (Aktion 1). Als Lösungskonzept verwenden Myatt und Wallace das des Nash-Gleichgewichts in reinen Strategien; im Fall mehrerer derartiger Gleichgewichte fordern sie zusätzlich Risikodominanz.[9]

Das erste Spiel entspricht im Kern dem von Johnson (2002) betrachteten, enthält also wiederum ein Trittbrettfahrerproblem. Beide Spieler erhalten als Benutzer die hier gleiche Auszahlung v, falls mindestens einer von ihnen ent-

[8] Alle Ergebnisse gelten aber für n Spieler ganz analog.

[9] Myatt und Wallace begründen ihre Wahl von Risikodominanz als Selektionskriterium damit, daß dieses Gleichgewichte wählt, die mit jenen eng verwandter Spiele mit unvollständigen Informationen korrespondieren.

Trittbrettfahrerproblem

	1	0
1	$v - c_1, v - c_2$	$v - c_1, v$
0	$v, v - c_2$	$0, 0$

(a) keine Integration

	1	0
1	$v - c_1/2, v - c_2/2$	$-c_1/2, 0$
0	$0, -c_2/2$	$0, 0$

(b) Integration

Problem positiver externer Effekte

	1	0
1	$2v - c_1, 2v - c_2$	$v - c_1, v$
0	$v, v - c_2$	$0, 0$

(c) keine Integration

	1	0
1	$2v - c_1, 2v - c_2$	$-c_1, 0$
0	$0, -c_2$	$0, 0$

(d) Integration

Abbildung 3.1: Von Myatt und Wallace (2002) analysierte Spiele

wickelt. Die Kosten dafür seien $c_1 > 0$ bzw. $c_2 > 0$; zudem gelte $c_1 < c_2$.[10] In Abbildung 3.1 (a) ist dieses Spiel in Normalform dargestellt.[11] Es gilt nun:

- Für $c_2 \leq v$ sind zwar die Strategieprofile $(1, 0)$ und $(0, 1)$ beide Nash-Gleichgewichte, aber nur $(1, 0)$ ist risikodominant; für $c_1 \leq v < c_2$ ist $(1, 0)$ sogar das einzige Gleichgewicht. In beiden Fällen entwickelt folglich ausschließlich der Spieler mit den geringeren Kosten!

- Für $v < c_1$ entwickelt hingegen niemand. Dieses Ergebnis ist aber für $c_1 < 2v$ ineffizient. Myatt und Wallace untersuchen nun, ob die Projektleitung diese Ineffizienz durch eine Änderung der Spielregeln beseitigen kann, indem sie die zu entwickelnde Erweiterung in zwei für sich genommen nutzlose Hälften zerlegt und jedem Spieler die Möglichkeit gibt, eine Hälfte zu entsprechend reduzierten Kosten zu entwickeln. Das Spiel nimmt dann die in Abbildung 3.1 (b) dargestellte Form an. Es zeigt sich aber, daß das Profil $(1, 1)$ unter der gegebenen Einschränkung $v < c_1 < 2v$ kein risikodominantes Gleichgewicht ist. Die Zerlegung der Erweiterung mit dem Zwang zur Integration ist also nicht zielführend.

[10] Der relativ uninteressante Fall $c_1 = c_2$ wird in dieser Darstellung ausgeklammert, da dann das Spiel symmetrisch wird und Risikodominanz als Selektionskriterium ausfällt.

[11] Wie immer ist Spieler 1 der Zeilenspieler und Spieler 2 der Spaltenspieler.

Des weiteren betrachten Myatt und Wallace eine Situation, in der jeder Spieler eine ihm genau zugeordnete Erweiterung entwickeln kann, die auch vom jeweils anderen Spieler geschätzt wird. Der Fokus dieser Modellvariante liegt also auf der Wirkung positiver externer Effekte. Das betrachtete Spiel findet sich in Abbildung 3.1 (c); die wichtigsten Ergebnisse sind nachfolgend angegeben:

- Für jeden Spieler $i \in \{1, 2\}$ ist es offensichtlich streng dominant beizutragen, wenn $v > c_i$ gilt. Anderenfalls ist es schwach dominant, nicht beizutragen.

- Das Verhalten der Spieler ist aufgrund der positiven externen Effekte nicht effizient. Anders als im ersten betrachteten Spiel kann die Projektleitung aber hier die Ineffizenz zumindest für bestimmte Parameterwerte beseitigen, indem sie beide Erweiterungen wiederum so integriert, daß sie keinen eigenständigen Nutzen haben. Es entsteht dann das in Abbildung 3.1 (d) gezeigte Spiel, welches für $c_2 \leq 2v$ die beiden Gleichgewichte $(1, 1)$ und $(0, 0)$ aufweist. Gilt $2v > c_1 + c_2$, dann ist $(1, 1)$ risikodominant und die Entwicklung erfolgt auf effizientem Niveau; für $2v < c_1 + c_2$ bricht sie hingegen vollkommen zusammen.

Das vorliegende Modell ist aus zwei Gründen bemerkenswert: Erstens zeigt es die Rolle asymmetrischen Verhaltens für die Lösung von Trittbrettfahrerproblemen, wenn gute Informationen über die beteiligten Entwickler vorliegen; zweitens verdeutlicht es die Bedeutung der Interaktionsstruktur zwischen den Spielern und ruft auch in Erinnerung, daß diese Struktur situativ verändert werden kann.

Schwachpunkte des Modells sind wiederum die Berücksichtigung nur eines Motivs sowie die artifizielle Einschränkung der Strategiemenge, welche es den Spielern nicht erlaubt, ihre Eigenentwicklungen zu verkaufen.

3.2.2 Die Rolle von Karrieremotiven

Die zweite Welle formaler Modelle hat sich dem von Lerner und Tirole (2002) gewiesenen Weg folgend der Rolle von Karrieremotiven zugewandt. Dabei wurden zwei grundsätzlich verschiedene Ansätze verfolgt: Einige Modelle erweitern den im vorherigen Abschnitt vorgestellten Rahmen der privaten Bereitstellung öffentlicher Güter dahingehend, daß nicht nur das Endergebnis, sondern auch das Entwickeln an sich einen eigenständigen Nutzen stiftet, der als Wert eines positiven Signals an potentielle Arbeitgeber oder neuerworbenen Könnens interpretiert werden kann. Andere Modelle blenden das Motiv der Befriedigung von Eigenbedarf hingegen vollkommen aus und betrachten reinrassiges Job-Market-Signaling à la Spence (1973), erweitert um Eigenarten offener Software. Nachfolgend wird für beide Ansätze jeweils ein repräsentatives Modell vorgestellt.

3.2.2.1 Das Modell von Bitzer und Schröder (2005)

Auch Bitzer und Schröder (2005) betrachten ein Spiel zwischen $n \in \mathbb{N}$ Spielern, in dem es um die Frage geht, wer eine von allen geschätzte Erweiterung entwickelt. Gegenüber den bisher vorgestellten Modellen verfügen die Spieler dabei aber über eine sehr viel elaboriertere Strategiemenge: Konnten die Spieler bislang nur sofort entwickeln oder aber nie, so ist eine reine Strategie in diesem Spiel, zum Zeitpunkt $t \in [0, \infty)$ zu entwickeln, sollte dies bis dahin noch nicht geschehen sein. Formal führt diese Modellierung zu einem Zermürbungskrieg in stetiger Zeit.

Jeder Spieler i ist durch sechs allgemein bekannte Parameter charakterisiert: den Nutzen $v_i \geq 0$, den er erhält, solange die Erweiterung noch nicht verfügbar ist, den Nutzen $u_i \geq v_i$, den er in seiner Rolle als Nutzer erhält, sobald die Erweiterung verfügbar ist, den Karrierenutzen $s_i \geq 0$, den ihm verbesserte Karrieremöglichkeiten stiften, wenn er selbst entwickelt, seine Entwicklungskosten $c_i \geq 0$, seine Diskontierungsrate $r_i \geq 0$ sowie den Zeithorizont $T_i \geq 0$, nach dessen Überschreitung der Spieler aus exogenen Gründen aus dem Spiel

ausscheidet. Dabei erfassen v_i, u_i und s_i jeweils die Größe eines Nutzenstroms, der zwischen zwei Zeitpunkten $t_1 \geq 0$ und $t_2 \geq t_1$ fließt und der z. B. für u_i den Gegenwartswert $\int_{t_1}^{t_2} u_i e^{-r_i t}\, dt = (u_i/r_i)(e^{-r_i t_1} - e^{-r_i t_2})$ aufweist.

Entwickelt nun zum Zeitpunkt t ein Spieler $j \neq i$ die Erweiterung, so erhält Spieler i den Nutzen $F_i(t) = (v_i/r_i)(1 - e^{-r_i t}) + (u_i/r_i)(e^{-r_i t} - e^{-r_i T_i})$; entwickelt hingegen Spieler i selbst, so stiftet ihm dies den Nutzen $D_i(t) = F_i(t) + (s_i/r_i)(e^{-r_i t} - e^{-r_i T_i}) - c_i e^{-r_i t}$. Entwickelt schließlich innerhalb des relevanten Zeithorizonts niemand die Erweiterung, so erhält Spieler i den Nutzen $R_i = (v_i/r_i)(1 - e^{-r_i T_i})$.

Bitzer und Schröder analysieren nun die Gleichgewichte dieses Spiels unter der zusätzlichen Annahme $(s_i/r_i)(1 - e^{-r_i T_i}) < c_i < ((s_i + u_i - v_i)/r_i)(1 - e^{-r_i T_i})$ für alle i. Wären nämlich die Entwicklungskosten c_i im Vergleich zum Karrierenutzen s_i zu klein, so würde Spieler i die Erweiterung trivialerweise sofort entwickeln, wären sie hingegen im Vergleich zum Karrierenutzen s_i und dem durch die Erweiterung gestifteten Zusatznutzen $u_i - v_i$ zu groß, so würde Spieler i unter keinen Umständen entwickeln und könnte in der Analyse vernachlässigt werden. Für entsprechend beschränkte Entwicklungskosten gilt nun:

- Für jeden Spieler i existiert ein kritischer Zeitpunkt $\bar{t}_i = T_i - (1/r_i) \cdot \ln\left((u_i - v_i + s_i)/(u_i - v_i + s_i - r_i c_i)\right)$, ab dem er unter keinen Umständen entwickelt. Dieser Zeitpunkt ergibt sich als Lösung von $D_i(t) = R_i$.

- Gilt $\bar{t}_i \neq \bar{t}_j$ für alle $i \neq j$, so existiert ein eindeutiges teilspielperfektes Gleichgewicht, in dem der Spieler mit dem größten $\bar{t}_i$ zum Zeitpunkt 0 entwickelt.

- Der die Erweiterung entwickelnde Spieler ist also tendenziell gekennzeichnet durch einen hohen Nutzenzugewinn $u_i - v_i$, einen hohen Karrierenutzen s_i, einen langen Zeithorizont T_i, eine niedrige Diskontierungsrate r_i und niedrige Kosten c_i.

Das Modell illustriert also, wie eine reichhaltige Strategiemenge das bisher betrachtete Koordinationsproblem zu lösen hilft. Die Relevanz dieses Aspekts

wird jedem Leser, der in seinem Leben bereits mit einer zweiten Person vor einem schmalen, nur allein passierbaren Durchgang stand, aber eben auch nach zunächst erfolglosen Koordinationsbemühungen nun nicht mehr dort steht, sofort einleuchten. Zudem benennt das Modell Eigenschaften, welche die Entwickler offener Software neben einfachen Nutzenaspekten tendenziell auszeichnen sollten: Jugend, Geduld und Können. Daß diese theoretisch hergeleiteten Eigenschaften den in Abschnitt 2.3 beschriebenen entsprechen, stützt das vorliegende Modell.

Wiederum ist es den Spielern allerdings nicht erlaubt, ihre Eigenentwicklungen zu verkaufen. Zwar bietet der Karrierenutzen immerhin einen formalen Grund dafür, die Eigenentwicklung als offene Software zu verschenken. Da dieser Nutzen aber annahmegemäß nach oben durch die Entwicklungskosten beschränkt ist, die wiederum regelmäßig niedriger sind als die mit einer Software am Markt erzielbaren Gewinne, sticht dieses Argument letztlich nicht.

3.2.2.2 Das Modell von Leppämäki und Mustonen (2004)

Leppämäki und Mustonen (2004) analysieren die Entwicklung offener Software in einem klassischen Job-Market-Signaling-Modell,[12] welches sie um die Besonderheit erweitern, daß die Signalisierungsaktivitäten Auswirkungen auf die Absatzmärkte potentieller Arbeitgeber haben. Die in dem Modell auftretenden Akteure sind ein Entwickler, ein Softwareunternehmen und mehrere Endnutzer, die anders als bisher keine potentiellen Entwickler sind.

Der Entwickler hat einen nur ihm bekannten Typ $\theta \in \{1, 2\}$,[13] der ein Maß für seine Fähigkeiten darstellt. Um nun dem Softwareunternehmen seine Fähigkeiten zu signalisieren, kann er eine offene Software der Qualität $y \in [0, \bar{y}]$ mit $\bar{y} > 0$ entwickeln; dabei entstehen dem Entwickler typabhängige Kosten in Höhe von

[12] Weitere Job-Market-Signaling-Modelle stammen von Lee, Moisa und Weiß (2004) sowie von Prüfer (2004).

[13] Allgemein bekannt sind lediglich die Wahrscheinlichkeiten $\Pr\{\theta = 2\} = q$ und $\Pr\{\theta = 1\} = 1 - q$ mit $q \in (0, 1)$.

y/θ. Schließlich zahlt ihm das Softwareunternehmen in Kenntnis der von ihm entwickelten offenen Software den Lohn $w \geq 0$. Der Gesamtnutzen des Entwicklers beträgt dann $u(w, y, \theta) = w - y/\theta$.

Die ursprünglich allein als Signal entwickelte offene Software hat nun auf dem Umweg über den Endnutzermarkt einen externen Effekt auf den zukünftigen Arbeitgeber, das Softwareunternehmen. Dieser Effekt ist positiv, wenn die offene Software und das Angebot des Softwareunternehmens Komplemente sind; im Fall von Substituten ist er hingegen negativ. Formal wird er durch den Parameter k erfaßt, der für $k > 0$ die Stärke der Komplementarität und für $k < 0$ die der Substituierbarkeit anzeigt. Unter der zusätzlichen Annahme, daß der Entwickler nach seiner Anstellung für das Unternehmen eine Software der Qualität θ entwickelt, beschreiben Leppämäki und Mustonen einen realistischen Endnutzermarkt der Größe $M > 0$, auf dem das Unternehmen den Umsatz $R = M \cdot (\theta + ky)$ erzielen kann. In Bezug auf den Arbeitsmarkt wird zudem angenommen, der Entwickler verfüge über alle Verhandlungsmacht; sein Lohn w entspricht dann bei Vernachlässigung sonstiger Kosten dem erwarteten Umsatz.

Die Zeitstruktur des Modells entspricht der anderer Signaling-Modelle; des weiteren fokussieren Leppämäki und Mustonen ihre Analyse auf das Perfekte Bayesianische Trenngleichgewicht mit den geringsten Signalisierungskosten, welches dem Intuitiven Kriterium genügt. In diesem Gleichgewicht gilt:[14]

- Offene Software wird nur von Entwicklern mit dem Typ $\theta = 2$ entwickelt. Ihre Qualität ist dann $y^*(\theta = 2) = M/(1 - Mk)$.
- In einer Modellerweiterung verfügt der Entwickler unabhängig vom Typ über eine alternative Anstellungsmöglichkeit mit dem Lohn $w_0 > M \cdot (1 + q)$. Gäbe es keine offene Software, so hätte das Unternehmen also einen negativen Erwartungsgewinn, wenn es den Entwickler ungeprüft zum Lohn w_0 einstellte. Folglich hat das Unternehmen einen Anreiz, ein offenes Entwicklungsprojekt zu begründen, welches als Screening-Mechanismus dienen kann.

[14] Die vorgestellten Ergebnisse gelten unter der technischen Annahme $k \leq 1/(2M)$.

Das vorliegende Modell illustriert also, wie Signaling-Erwägungen kompetente Einzelpersonen zur Entwicklung offener Software bewegen können. Die Berücksichtigung der geschilderten externen Effekte erlaubt überdies Aussagen darüber, wie hoch die Qualität der aus diesem Antrieb geschaffenen offenen Software ist: sie sollte sowohl mit der Marktgröße als auch mit dem Grad der Komplementarität zwischen offener und proprietärer Software ansteigen. Des weiteren zeigt das Modell, daß es für ein Unternehmen sinnvoll sein kann, ein offenes Entwicklungsprojekt zu begründen, wenn es so die Möglichkeit zum Screening potentieller Arbeitnehmer erhält.

Leider greift die alleinige Berücksichtigung des Signaling-Motivs wie gesehen zu kurz. Des weiteren bleibt unklar, welche Resultate sich ergäben, wenn mehrere Entwickler und Unternehmen sowohl auf dem Arbeits- als auch auf dem Endnutzermarkt konkurrierten.

3.2.3 Die Rolle von Unternehmen

In jüngster Zeit wurden verschiedene theoretische Arbeiten zu den Motiven von Unternehmen veröffentlicht. Kein dem Verfasser dieser Arbeit bekanntes Modell befaßt sich dabei explizit mit solchen Unternehmen, die zu einer offenen Software komplementärere Produkte anbieten. Eine Lücke hinterläßt das Fehlen einer solchen Betrachtung allerdings nicht, da die Koordinationsprobleme zwischen mehreren Anbietern von Komplementen mit den in Abschnitt 3.2.1 beschriebenen identisch sind; spezielle Modelle sind daher nicht erforderlich. Nachfolgend werden allerdings zwei Modelle vorgestellt, die weitere Gründe dafür offenlegen, warum Unternehmen zur Entwicklung offener Software beitragen.[15]

3.2.3.1 Das Modell von Mustonen (2005)

Mustonen (2005) untersucht, warum sich ein Unternehmen an der Entwicklung einer offenen Software beteiligen könnte, obwohl diese ein *Substitut* zu einem

[15] Auch Leppämäki und Mustonen (2004) benennen wie gesehen mit dem Screening potentieller Mitarbeiter ein Motiv für die Beteiligung an offener Software.

eigenen Produkt darstellt.

Dazu betrachtet er einen Markt, in dem ein Kontinuum von Konsumenten der Masse 1 zwischen einer offenen und einer proprietären Software wählen kann. Jeder Konsument wird charakterisiert durch seinen Typ θ, der auf dem Intervall $[0,1]$ gleichverteilt sei. Dieser Typ entspricht dem Grundnutzen, den der Konsument aus der offenen Software bezieht; des weiteren wird angenommen, daß die proprietäre Software hochwertiger ist und der entsprechende Grundnutzen 2θ beträgt. Zusätzlich zum Grundnutzen stiften beide Angebote einen Netzwerknutzen.[16] Bezeichnet x den *erwarteten* Marktanteil der proprietären Software, $\kappa \in \{0,1\}$ den Kompatibilitätsgrad zwischen den beiden Angeboten und $v \geq 0$ ein Maß für die Stärke des Netzwerkeffekts, dann betrage diese Nutzenkomponente für die proprietäre Software $v \cdot (x + \kappa \cdot (1-x))$ und für die offene $v \cdot (1-x+\kappa x)$.[17] Des weiteren wird angenommen, die offene Entwicklungsgemeinschaft stifte einen zusätzlichen Nutzen in Höhe von $\beta \geq 0$, auf den bei Inkompatibilität nur die Nutzer der offenen Software, bei Kompatibilität hingegen alle Nutzer Zugriff haben. Schließlich erhebe der Hersteller der proprietären Software Lizengebühren in Höhe von $l \geq 0$. Ein Konsument von Typ θ bezieht also aus der proprietären Software den Nutzen $u_P(\theta, l, \kappa) = 2\theta + v \cdot (x + \kappa \cdot (1-x)) + \kappa\beta - l$ und aus der offenen den Nutzen $u_O(\theta, \kappa) = \theta + v \cdot (1 - x + \kappa x) + \beta$.

Mustonen betrachtet wie in der Literatur zu Netzwerkeffekten üblich nur Nash-Gleichgewichte, in denen die erwarteten Marktanteile den tatsächlichen entsprechen. Unter den beiden Annahmen $v < 1/3$ und $\beta < 1 - v$ gilt dann:[18]

- Bei Kompatibilität beträgt der Unternehmensgewinn $G^*_{\kappa=1} = 1/4$, bei Inkompatibilität hingegen $G^*_{\kappa=0} = (1 - v - \beta)^2/(4 - 8v)$.

[16] In Abschnitt 4.2.1 wird die Rolle von Netzwerkeffekten in Softwaremärkten ausführlich beschrieben.

[17] In dem Modell erwirbt jeder Konsument genau eines der beiden Angebote.

[18] Tatsächlich analysiert Mustonen sein Modell für einen größeren Parameterraum, übersieht dabei aber, daß für starke Netzwerkeffekte (großes v) aufgrund des Tipping-Effekts mehrere Gleichgewichte existieren. Der Haupteffekt seines Modells läßt sich aber auch in dem hier betrachteten, eingeschränkten Parameterraum illustrieren.

- Für $\beta > 1 - v - \sqrt{1-2v}$ gilt $G^*_{\kappa=1} > G^*_{\kappa=0}$. Das Unternehmen hat also einen Anreiz, Kompatibilität herzustellen, wenn der von der offenen Entwicklungsgemeinschaft gestiftete Zusatznutzen β hinreichend groß ist.

Ist nun Kompatibilität am kostengünstigsten durch eine Veränderung der offenen Software herzustellen und übersteigt die Differenz $G^*_{\kappa=1} - G^*_{\kappa=0}$ diese Kosten, so wird sich das Unternehmen mit dem entsprechenden Ziel an der Weiterentwicklung der substitutiven offenen Software beteiligen. Es ist das Verdienst des vorliegenden Modells, diesen Effekt erstmalig beschrieben zu haben.

Leider krankt das Modell aber an zwei fragwürdigen Annahmen: Erstens ist zweifelhaft, ob die für das obige Ergebnis unverzichtbare Annahme, Nutzer einer proprietären Software könnten von Leistungen der Entwicklergemeinschaft einer kompatiblen offenen Software profitieren, der Realität entspricht; eine überzeugende Begründung gibt Mustonen jedenfalls nicht. Überdies ist zweitens die Annahme vertikaler Produktdifferenzierung wenig allgemeingültig. Sicherlich existieren Beispiele für Märkte, in denen eine proprietäre einer offenen Software qualitativ überlegen ist. Für ein generisches Modell, welches den Wettbewerb zwischen diesen beiden Arten von Software grundsätzlich beschreibt, scheint aber im Licht von Abschnitt 2.5 die Annahme horizontaler Differenzierung angemessener zu sein.

Tatsächlich ist es aber leicht, das von Mustonen beschriebene Motiv auf der Grundlage der in Abschnitt 4.2 beschriebenen Modellfamilie auch ohne diese beiden fragwürdigen Annahmen zu replizieren: In Abschnitt 4.2.2 wird der Wettbewerb zwischen einer offenen und einer proprietären Software unter der Annahme horizontaler Differenzierung und ohne Netzwerkeffekte modelliert. Da das Fehlen von Netzwerkeffekten formal mit dem Fall vollkommener Kompatibilität identisch ist, kann man den dort hergeleiteten Unternehmensgewinn hier als Gewinn des Unternehmens bei Kompatibilität interpretieren; er ist demnach $G'_{\kappa=1} = t/8$ mit $t \geq$ als Präferenzintensität bezüglich der horizontalen Differenzierung.[19] In Abschnitt 4.2.3 wird dann gezeigt, daß der Gewinn des

[19]Der Gewinn ergibt sich, indem man in Gleichung (4.7) die beiden hier nicht relevanten

Unternehmens bei Inkompatibilität und schwachem Netzwerkeffekt ($v < t$) nur $G'_{\kappa=0} = (t - v)/8$ beträgt. Der oben beschriebene Grund für die Beteiligung an einem offenen Entwicklungsprojekt ist also wiederum gültig.

3.2.3.2 Das Modell von Henkel (2004b)

Henkel (2004b) betrachtet ein Modell mit zwei Unternehmen A und B, deren Produkte in einem Qualitätswettbewerb stehen. Beide Unternehmen verwenden in ihrem jeweiligen Produkt eine offene Software, die um zwei Module mit frei wählbarer Qualität erweitert werden kann. Dabei ist Modul 1 für Unternehmen A und Modul 2 für Unternehmen B relativ wichtiger. Bezeichnet q_{Xi} das Qualitätsniveau, mit dem Modul $i \in \{1, 2\}$ Unternehmen $X \in \{A, B\}$ zur Verfügung steht, und ist $a \in [0, 1]$ ein Maß für die Homogenität des Bedarfs der beiden Unternehmen bezüglich der zwei Module, dann sei die Gesamtqualität der Produkte von A und B gegeben durch $Q_A = q_{A1} + aq_{A2} + q_{A1}q_{A2}$ bzw. $Q_B = aq_{B1} + q_{B2} + q_{B1}q_{B2}$. Der Term $q_{X1}q_{X2}$ drückt dabei aus, daß es sich um zwei komplementäre Module handelt.

Unternehmen X kann in die Entwicklung von Modul i den Betrag c_{Xi} investieren und so das Qualitätsniveau $\sqrt{c_{Xi}}$ erzeugen. Des weiteren kann es diese Entwicklung allgemein verfügbar machen ($g_{Xi} = 1$) oder aber geheimhalten ($g_{Xi} = 0$). Das von Unternehmen X letztendlich verwendete Modul i hat dann die Qualität $q_{Xi} = \max\{\sqrt{c_{Xi}}, g_{Yi} \cdot \sqrt{c_{Yi}}\}$ mit $Y \neq X$. Die Gesamtkosten betragen $C_X = c_{X1} + c_{X2}$.

Henkel verzichtet auf eine explizite Modellierung des Qualitätswettbewerbs. Vereinfachend nimmt er an, der Gewinn der Unternehmen A und B betrage $G_A = Q_A - cQ_B - C_A$ bzw. $G_B = Q_B - cQ_A - C_B$. Dabei ist $c \in [0, 1]$ ein Maß für die Intensität des Wettbewerbs zwischen den beiden Unternehmen.

Die Entscheidungen fallen in der folgenden Reihenfolge: Erst entscheiden beide Unternehmen simultan über ihren Markteintritt. Dann legen sie verbindlich fest,

Politikvariablen z und s gleich Null setzt.

ob sie ihre Eigenentwicklungen veröffentlichen oder aber geheimhalten. Schließlich treffen sie ihre Entwicklungsentscheidung. Das geeignete Lösungskonzept ist offensichtlich das teilspielperfekte Gleichgewicht. Es gilt nun:

- Für verhältnismäßig große Werte von a und c (z. B. für $a = 0,8$ und $c = 0,7$)[20] existiert ein eindeutiges Gleichgewicht, in dem beide Unternehmen in den Markt eintreten, ihre Entwicklungen veröffentlichen und nur das für sie wichtigere Modul entwickeln. Dieses Gleichgewicht beruht im Kern darauf, daß es für die gegebenen Parameterwerte schlicht zu teuer ist, beide Module zu entwickeln.

- Für verhältnismäßig kleine Werte von a und c (z. B. für $a = 0,2$ und $c = 0,2$) existiert zusätzlich zu dem eben beschriebenen kooperativen Gleichgewicht ein weiteres, in dem beide Unternehmen in den Markt eintreten und ihre Entwicklungen geheimhalten. Das unerwartete kooperative Gleichgewicht begründet sich dabei wie folgt: Schert etwa Unternehmen B einseitig aus der Entwicklungsgemeinschaft aus, so kann Unternehmen A das von Unternehmen B in hoher Qualität entwickelte Modul 2 nicht mehr verwenden. Es wird also Modul 2 fortan in Eigenregie entwickeln, allerdings zu einer deutlich geringeren Qualität, da ihm Modul 2 für ein kleines a sehr unwichtig ist. Da aber die beiden Module Komplemente sind, verringert sich für Unternehmen A dann auch der Anreiz, Modul 1 in einer hohen Qualität zu entwickeln. Unter diesem Qualitätsverlust leidet auch Unternehmen B, welches Modul 1 ja von Unternehmen A übernimmt. Folglich sinken durch den Wechsel des Unternehmens B zu einer Strategie der Geheimhaltung sowohl Q_A als auch Q_B; ist nun die Wettbewerbsintensität c hinreichend klein, reduziert dies auch den Gewinn G_B.

Das vorliegende Modell zeigt ganz ausgezeichnet, wie zwei Unternehmen unter bestimmten Rahmenbedingungen eine für beide Seiten profitable Entwickler-

[20] Die genaue algebraische Beschreibung der erforderlichen Werte der Parameter a und c ist umfangreich und kaum informativ; auf sie wird daher hier verzichtet.

gemeinschaft bilden können. Insbesondere der zuletzt beschriebene Fall zweier Gleichgewichte ist dabei interessant: Im unkooperativen Gleichgewicht halten beide Unternehmen ihre Eigenentwicklungen geheim; im kooperativen, Paretodominanten Gleichgewicht entwickelt hingegen jedes Unternehmen das ihm wichtigere Modul und stellt es dem anderen als offene Software zur Verfügung. Die Unternehmen spielen also ein Koordinationsspiel.

Allerdings behandelt das Modell einen zu kleinen Ausschnitt der Wirklichkeit, um grundsätzliche Aussagen über die Arbeitsweise offener Softwareprojekte zu ermöglichen. Zudem bleibt unklar, wieso die Unternehmen ihre Eigenentwicklungen nicht wechselseitig lizenzieren.

3.3 Modell zur Entstehung offener Softwareprojekte

3.3.1 Grundidee

Die im vorigen Abschnitt vorgestellten Modelle illustrieren, wie verschiedenste Motive ein offenes Softwareprojekt tragen, sobald dieses als Institution etabliert ist. Sie erklären allerdings nicht, warum und unter welchen Voraussetzungen ein offenes Softwareprojekt überhaupt erst entsteht. So nehmen Johnson (2002), Myatt und Wallace (2002) sowie Bitzer und Schröder (2005) den Entwicklern allesamt die Option, Eigenentwicklungen zu verkaufen; gerechtfertigt ist diese Einschränkung aber eben nur dann, wenn diese Eigenentwicklung auf einer bereits vorhandenen unter einer Copyleft-Lizenz stehenden offenen Software aufbaut. Auch Leppämäki und Mustonen (2004) setzen das Vorhandensein einer hierarchischen Entwicklergemeinschaft mit etablierten Begutachtungsmechanismen implizit voraus. Schließlich erfordert Job-Market-Signaling eine anerkannte Institution, die den Erwerb eines Signals für talentierte Individuen billig und für untalentierte teuer macht, und es einem Arbeitgeber so erspart, die Fähigkeiten eines potentiellen Arbeitnehmers selbst ermitteln zu müssen. Und Mustonen (2005) und Henkel (2004b) nehmen in ihren Modellen sogar die Existenz einer reifen, massenmarkttauglichen offenen Software an.

Um die in der bestehenden Literatur vorhandene Lücke zu füllen, wird in dieser Arbeit ein Modell entwickelt, welches die Entstehung offener Softwareprojekte erklärt. Das Modell beruht auf dem evolutionären Ansatz zur Entstehung von Institutionen von Kandori, Mailath und Rob (1993) sowie Young (1993). Diese postulieren eine Welt, in der Institutionen erst im Laufe der Zeit durch die wiederholte Interaktion vieler beschränkt rationaler Individuen entstehen, anstatt sich wie sonst üblich als Gleichgewichte von Spielen zwischen perfekt rationalen homines oeconomici sofort zu etablieren.[21] Relevant ist dieser Unterschied deshalb, da er zu abweichenden Schlußfolgerungen führen kann. So schreibt Young (1998, S. 4):

> Neoclassical economics describes the way the world looks once the dust has settled; we are interested in how the dust goes about settling. This is not an idle issue, since the business of settling may have considerable bearing on how things look afterwards.

Das hier entwickelte Modell fußt ganz wesentlich auf Vorarbeiten von Eshel, Samuelson und Shaked (1998), die den oben beschriebenen evolutionären Ansatz auf ein Modell zur privaten Bereitstellung öffentlicher Güter anwenden. Zwei Eigenschaften ihres Modells verdienen dabei besondere Aufmerksamkeit: Erstens analysieren sie ein *lokales* Gut, d. h. nicht alle profitieren vom Beitrag eines bestimmten Spielers, sondern nur jene in einer geeignet definierten Nachbarschaft dieses Spielers. Zweitens treffen alle Spieler ihre Entscheidungen, indem sie besonders erfolgreiche Spieler in ihrer Nachbarschaft imitieren; sie sind also ausgesprochen beschränkt. Eshel et al. (1998) zeigen, daß unter bestimmten Voraussetzungen fast alle Spieler zum öffentlichen Gut beitragen.

Um verschiedene wichtige Spezifika offener Software angemessen berücksichtigen

[21] Den Hauptgrund dafür, die Annahme perfekt rationaler Spieler abzulehnen, liefert die experimentelle Spieltheorie. Sie hat auf vielfältige Weise belegt, daß sich echte Menschen nur sehr selten vollkommen rational verhalten. Eine gute Einführung in beschränkte Rationalität und ökonomisches Verhalten gibt Selten (1991).

zu können, wird das Modell von Eshel et al. (1998) in mehrfacher Hinsicht modifziert:

- Erstens liegt dem hier entwickelten Modell eine andere Auszahlungsstruktur zugrunde, da es zwei Nutzenkomponenten berücksichtigt, die nur bei offener Software, nicht aber bei öffentlichen Gütern allgemein vorhanden sind: So ist die Verfügbarkeit des Quelltexts wertvoll an sich; beispielsweise kann man nur durch Einsicht in den Quelltext einer Software deren Korrektheit und Sicherheit umfassend ermitteln, sie an eigene Bedürfnisse anpassen und von ihr lernen. Ceteris paribus ist also Software gerade für technisch versierte Nutzer sehr viel wertvoller, wenn sie als öffentliches und nicht als privates Gut vorliegt.[22] Zudem birgt offene Software einige Nutzenkomponenten, auf die nur jene Zugriff erhalten, welche die Software als öffentliches Gut bereitstellen. Diese Nutzenkomponenten wurden in Abschnitt 3.1 bereits umfassend diskutiert; bei aller Verschiedenheit ist ihr einigendes Band, daß sie eine Entwicklergemeinschaft einer gewissen Größe erfordern.

- Zweitens wird die von Eshel et al. (1998) betrachtete Nachbarschaftsstruktur verallgemeinert. Dieser Schritt ist notwendig, da diese Struktur einen wichtigen Aspekt offener Softwareentwicklung beschreibt, der ohne die Veränderung zu starr erfaßt würde.

- Drittens wird schließlich das Entscheidungsverhalten der Spieler reichhaltiger modelliert als bei Eshel et al. (1998), um genauere Einsichten in den Zusammenhang zwischen den Verhaltensannahmen und den Modellergebnissen zu gewinnen.

Anders als die in Abschnitt 3.2 vorgestellten Arbeiten betrachtet das hier entwickelte Modell ganz gezielt die Entscheidung dazwischen, eine selbst entwickelte Software unter einer offenen Lizenz zu verschenken oder unter einer proprietären Lizenz zu verkaufen. Die vorgelagerte Frage, ob die Software überhaupt

[22] Raymond (2001, S. 142) erläutert den Wert offenliegender Quelltexte ausführlich.

entwickelt werden soll, wird hingegen der Einfachheit halber nicht betrachtet. Das Modell nimmt also implizit an, die individuell zu tragenden Entwicklungskosten seien im Verhältnis zur Zahlungsbereitschaft vernachlässigbar gering. Angemessen ist diese Vereinfachung aus drei Gründen: Erstens werden viele Entwickler – wie in Abschnitt 3.1 gezeigt – recht stark von intrinsischen Motiven angetrieben. Wenn auch diese Motive in vielen Fällen nicht ausreichen, um die Beteiligung an offenen Entwicklergemeinschaften nachhaltig zu sichern, so werden sie doch die individuellen Kosten dauerhaft reduzieren. Zweitens verfügen viele Entwickler – wie in Abschnitt 2.3 beschrieben – über eine universitäre Informatikausbildung. Da für die Entwicklung offener Software regelmäßig dieselben frei verfügbaren Werkzeuge verwendet werden wie in der universitären Lehre, finden viele Entwickler in offenen Softwareprojekten ein vertrautes Arbeitsumfeld vor; Lerner und Tirole (2002) sprechen in diesem Zusammenhang vom Alumni-Effekt. Drittens sorgt die in Abschnitt 2.4.3 beschriebene Modularisierung dafür, daß der einzelne nur einen kleinen Teil der gesamten Entwicklungslast zu tragen hat.

Das Modell berücksichtigt den Aspekt der Modularisierung aber auch explizit und behandelt in der Folge ein Koordinationsproblem, welches in subtiler Weise von dem in den bisher vorgestellten Modellen abweicht. In jenen Modellen erwächst der Koordinationsbedarf aus der Frage, wer von mehreren potentiellen Entwicklern ein ganz bestimmtes Modul verbessert; die Beiträge der Spieler sind also Substitute. In Abschnitt 2.4.3 wurde aber erläutert, daß an einem einzelnen Modul in der Realität nur sehr wenige Entwickler arbeiten (Durchschnitt 4, Median 1). Der entsprechende Koordinationsbedarf ist daher entweder gar nicht vorhanden oder aber, da eine geringe Spielerzahl die Annahme guter Informationen rechtfertigt, wie von Myatt und Wallace (2002) und Bitzer und Schröder (2005) gezeigt, leicht zu befriedigen. Im hier entwickelten Modell besteht das Koordinationsproblem hingegen darin, ob Einzelentwickler bzw. kleine Teams ihre Eigenentwicklungen proprietär vermarkten oder aber gemeinsam zu einer großen offenen Software wie etwa GNU/Linux verschmelzen; die Beiträge der Spieler sind also Komplemente.

3.3.2 Modellbeschreibung

3.3.2.1 Stufenspiel

Betrachtet wird eine endliche Menge $I = \{1, \ldots, n\}$ homogener Spieler, die wiederum als Nutzer und als Entwickler fungieren. Im Stufenspiel wählt nun jeder Spieler i eine Aktion $a_i \in \{0, 1\}$; $a_i = 0$ repräsentiert die Entscheidung, die Eigenentwicklung unter einer proprietären Lizenz zu verkaufen, für $a_i = 1$ verschenkt Spieler i sein Modul hingegen unter einer offenen Lizenz. Jeder Spieler ist dabei genau einem Modul fest zugeordnet. Die Aktionen aller Spieler werden im Aktionsprofil $a = (a_1, \ldots, a_n) \in A = \{0, 1\}^n$ zusammengefaßt.

Um die Beziehung der einzelnen Module zueinander zu erfassen, wird angenommen, sie seien gleichförmig entlang eines Kreises angeordnet. Eine solche Anordnung findet sich auch bei Bergstrom und Stark (1993), Ellison (1993) sowie Eshel et al. (1998); ihr Vorteil besteht darin, daß sie keine Asymmetrien erzeugt, wie sie etwa im Fall einer Geraden an den Rändern entstünden. Jeder Spieler i sei nun in eine Nachbarschaft $N(i) = \{i - r, \ldots, i + r\} \subseteq I$ mit dem Radius $r \geq 1$ eingebettet (vgl. Abbildung 3.2);[23] es gelte $n \geq 2r + 1$, damit sich die beiden Seiten einer Nachbarschaft nicht überlappen.

Wie bereits angedeutet, erfaßt die Nachbarschaftsstruktur keine räumlichen Beziehungen. Vielmehr beschreibt sie die Tatsache, daß die Module aus Nutzersicht einen unterschiedlich engen Bezug haben; so weisen etwa im Fall eines Betriebssystems wie GNU/Linux ein Treiber für eine Netzwerkkarte und die Implementierung eines Netzwerkprotokolls eine größere Nähe auf als eine Grafikbibliothek und ein Dateisystem. Diese Interpretation der Nachbarschaftsstruktur erlaubt die Annahme, alle Spieler in $N(i)$ hätten eine positive, auf 1 normierte Zahlungsbereitschaft für das von Spieler i entwickelte Modul, wohingegen alle anderen Spieler dieses Modul nicht wertschätzen. Wenn also Spieler i entscheidet, sein Modul zu verkaufen, so wird er einen Preis in Höhe von 1 erheben und den

[23] Im Interesse einer übersichtlichen Notation wird bei Spieler-Indizes $i + r$ für $[(i + r - 1) \text{ modulo } n] + 1$ und $i - r$ für $[(i - r + n - 1) \text{ modulo } n] + 1$ geschrieben. Beträgt also beispielsweise $n = 100$, dann gilt $100 + 1 = 1$ und $1 - 1 = 100$.

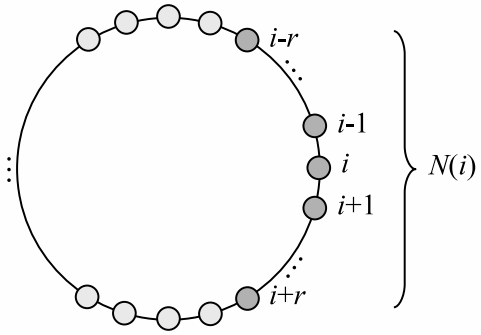


Abbildung 3.2: Anordnung der Module und Nachbarschaftsstruktur

Umsatz $2r$ erzielen.[24] Seine Kunden erhalten hingegen keinen positiven Nutzen, da ihre gesamte Zahlungsbereitschaft abgeschöpft wird. Wenn sich Spieler i hingegen dazu entschließt, sein Modul zu verschenken, erzielt er selbst keinen Umsatz. Die anderen Spieler in seiner Nachbarschaft profitieren hingegen in doppelter Hinsicht: Nicht nur können sie das Modul in binärer Form kostenlos verwenden, sondern sie haben zudem Zugriff auf den Quelltext. Jedem Spieler in $N(i)$ stifte Modul i daher den Nutzen $1 + v$, wobei $v \geq 0$ den Wert des Quelltexts erfaßt.

Um die in Abschnitt 3.1 beschriebenen Motive zu berücksichtigen, erhalten alle Spieler, die sich an der Entwicklergemeinschaft beteiligen, exklusiv die zusätzliche Auszahlung $s \geq 0$, wenn die Größe der Entwicklergemeinschaft $g(a) = \sum_{i=1}^{n} a_i$ die kritische Schranke $\hat{n} > 1$ übersteigt. Diese Schranke ergibt sich dabei als Anteil $\zeta \in (0, 1)$ von n gemäß $\hat{n} = \lfloor \zeta n \rfloor$;[25] ζ ist also ein Maß dafür, wie schwierig es ist, eine funktionierende offene Entwicklergemeinschaft zu errichten. Zusammengefaßt stiftet die Mitarbeit in der Entwicklergemeinschaft also den Nutzen

$$\sigma(a) = \begin{cases} s & \text{wenn } g(a) \geq \hat{n} \\ 0 & \text{sonst} \end{cases} . \tag{3.1}$$

[24] Der Einfachheit halber wird angenommen, indifferente Spieler kauften das Modul.

[25] Die Schranke $\hat{n}$ ergibt sich demnach als größte natürliche Zahl mit $\hat{n} \leq \zeta n$.

Die Nutzenfunktion jedes Spielers i lautet demnach

$$u_i(a) = (1 - a_i) \cdot 2r + a_i \cdot \sigma(a) + (1 + v) \cdot \sum_{j \in N(i) \setminus \{i\}} a_j. \tag{3.2}$$

Dabei erfaßt der Term $(1 - a_i)2r$ den durch den Verkauf des Moduls erzielbaren Umsatz, der Term $a_i\sigma(a)$ den Nutzen aus der Beteiligung an der Entwicklergemeinschaft und der Term $(1 + v)\sum a_j$ den Nutzen aus den als offene Software verfügbaren Modulen anderer. Man beachte, daß die letzte Nutzenkomponente unabhängig von der Aktion des Spielers i ist. Sie wird sich dennoch als sehr wichtig erweisen, sobald imitationales Verhalten analysiert wird. Des weiteren beachte man, daß u_i weder den Nutzen aus dem selbst entwickelten Modul noch die Entwicklungskosten enthalten muß, da jeder Spieler aufgrund der Annahmen immer entwickelt.

Wird dieses Spiel von perfekt rationalen Spielern bei simultaner Wahl der Aktionen genau einmal gespielt, gilt es zwei Fälle zu unterscheiden: Für $s < 2r$ ist der Verkauf von Modulen eine streng dominante Strategie und somit das Aktionsprofil $z^0 = (0, \ldots, 0) \in A$ das einzige Nash-Gleichgewicht. Dieses Gleichgewicht wird allerdings vom Aktionsprofil $z^1 = (1, \ldots, 1) \in A$ Pareto-dominiert. Alle Spieler wären bessergestellt, wenn jeder sein Modul als offene Software verschenkte, da die Quelltexte zusätzlichen Nutzen stiften. Die Spieler stehen also vor einem klassischen sozialen Dilemma. Für $s \geq 2r$ sind hingegen sowohl z^0 als auch z^1 Nash-Gleichgewichte. Das Spiel wird zu einem klassichen Koordinationsspiel, dessen Gleichgewichte nach dem Pareto-Kriterium geordnet werden können. Der Parameter ζ kann in diesem Fall als Maß für die *strategische Unsicherheit* interpretiert werden.

3.3.2.2 Dynamik

Das Ziel des Modells ist es zu erklären, warum und unter welchen Voraussetzungen offene Softwareprojekte entstehen. Eine offensichtliche Antwort auf die erste Frage ergäbe sich aus der Annahme perfekt rationaler Spieler und einer unendlichen Wiederholung des Spiels. Die geeignete Variante des Folk-Theorems

für wiederholte Spiele besagte dann, daß ein teilspielperfektes Gleichgewicht existiert, in dem alle Spieler in jeder Periode offene Software entwickeln.[26] Dieses Vorgehen ist aber für die hier betrachtete Fragestellung aus mehreren Gründen unangemessen. Denn die Anwendung des Folk-Theorems erzeugt nicht nur das oben beschriebene Gleichgewicht, sondern unendlich viele weitere einschließlich eines solchen, in dem niemals offene Software entwickelt wird. Es hilft daher nicht, die zweite, für die Praxis wichtigere Frage zu beantworten, welche Voraussetzungen die Entstehung einer offenen Entwicklergemeinschaft begünstigen. Zudem beruhen viele vom Folk-Theorem beschriebene Gleichgewichte auf äußerst sophistizierten Strategien und einem starkem Vertrauen in die unendliche Wiederholung des Spiels; gerade in Fällen mit sehr vielen Spielern werden die aus dem Folk-Theorem abgeleiteten Aussagen der Realität daher kaum gerecht. In dieser Arbeit wird nun ein anderer Ansatz gewählt, der die Selektion genau eines Gleichgewichts ermöglicht und so präzisere Antworten auf *beide* oben aufgeworfene Fragen gibt.

Auch in diesem Ansatz wird das oben beschriebene Stufenspiel in diskreten Zeitperioden $t \in \{0, 1, 2, \ldots\}$ unendlich oft wiederholt;[27] $a^t = (a_1^t, \ldots, a_n^t)$ bezeichnet das Aktionsprofil aller Agenten in Periode t. Formal wird nun ein homogener Markow-Prozeß (A, P) mit der Menge aller möglichen Aktionsprofile A als Zustandsraum konstruiert.

Die Wahrscheinlichkeiten $P = (p_{aa'})_{A \times A}$ für den Übergang von einem Zustand a zu einem Zustand a' resultieren aus dem beschränkt rationalen Verhalten der Spieler. In jeder Periode verwendet jeder Spieler i dabei eine von $j = 1, \ldots, m$ Entscheidungsregeln mit einer Wahrscheinlichkeit $\mu_j^i > 0$. Jede Entscheidungsregel d_j ist eine Funktion, die den aktuellen Zustand a und die Identität des

[26] Die für das gegebene Spiel passende Variante des Folk-Theorems beschreiben beispielsweise Fudenberg und Tirole (1991, S. 150).

[27] Die Annahme wiederholter und friktionsfreier Interaktion ist nur zulässig, wenn die Spieler über einen kostengünstigen und schnellen Kommunikationsmechanismus verfügen. In der Realität offener Softwareentwicklung ist dieser Mechanismus wie gesehen das Internet.

Spielers i auf die Aktion in der nächsten Periode abbildet, formal gilt also

$$a_i^{t+1} = d_j(a^t, i). \tag{3.3}$$

Ein einfaches Beispiel für eine solche Entscheidungsregel ist, in der Folgeperiode dieselbe Aktion wie in der aktuellen Periode zu verwenden. Diese Regel wird nachfolgend *Trägheitsregel* genannt und mit $\mathcal{T}(a, i) = a_i$ bezeichnet.

Die Wahrscheinlichkeit $\bar{p}_i(a_i^{t+1}|a^t)$, mit der Spieler i im Zustand a^t für die Folgeperiode die Aktion a_i^{t+1} wählt, ist dann gegeben durch

$$\bar{p}_i(1|a^t) = \sum_{j=1}^{m} \mu_j^i d_j(a^t, i) \quad \text{und} \quad \bar{p}_i(0|a^t) = \sum_{j=1}^{m} \mu_j^i (1 - d_j(a^t, i)). \tag{3.4}$$

Unter der Annahme, daß alle Spieler ihre Entscheidung voneinander unabhängig treffen, beträgt die Übergangswahrscheinlichkeit von Zustand a zu Zustand a'

$$p_{aa'} = \prod_{i=1}^{n} \bar{p}_i(a_i'|a). \tag{3.5}$$

Der so spezifizierte Markow-Prozeß wird nachfolgend mithilfe der folgenden Konzepte analysiert: Eine *Rekurrenzklasse* ist eine Menge von Zuständen $Z \subseteq A$ mit den zwei Eigenschaften, daß die Wahrscheinlichkeit, von einem Zustand innerhalb von Z zu einem Zustand außerhalb von Z zu wechseln, Null beträgt und daß die Wahrscheinlichkeit, innerhalb endlich vieler Schritte von jedem Zustand in Z zu jedem anderen Zustand in Z zu wechseln, echt positiv ist. Ein Zustand z, der eine einelementige Rekurrenzklasse $\{z\}$ bildet, heißt *absorbierend*; ein Zustand, der in keiner Rekurrenzklasse enthalten ist, heißt *transient*. In der Tradition von Kandori, Mailath und Rob (1993) und Young (1993) werden Rekurrenzklassen und absorbierende Zustände im folgenden als kurzfristige Zustandsvorhersagen des Modells interpretiert. Man beachte, daß üblicherweise mehrere Rekurrenzklassen existieren und daß die Wahrscheinlichkeit, mit der ein Markow-Prozeß in eine Rekurrenzklasse findet, vom Startzustand a^0 abhängt. Die kurzfristigen Vorhersagen sind dementsprechend vage.

3.3.2.3 Perturbierte Dynamik

Eine plausible langfristige Modellvorhersage sollte sich zudem durch eine besondere Stabilität gegenüber stochastischen Störungen auszeichnen. Daher wird die oben beschriebene Dynamik nachfolgend um stochastisches Rauschen erweitert. Es wird angenommen, jeder Spieler i würde sich nur mit der Wahrscheinlichkeit $1-\varepsilon$ einer der Entscheidungsregeln bedienen, und mit einer kleinen Wahrscheinlichkeit $\varepsilon > 0$ seine Aktion zufällig wählen. Die zufällige Wahl der Aktion wird nachfolgend *Fehler* genannt. Die Berücksichtigung von Fehlern ist aus zwei Gründen sinnvoll: Erstens erscheint es realistisch anzunehmen, daß Menschen bei der Anwendung von Entscheidungsregeln gelegentlich Fehler machen, sei es, da sie über fehlerhafte Daten verfügen oder da sie korrekte Daten falsch verarbeiten. Zweitens repräsentieren Fehler all jene entscheidungsrelevanten Einflüsse, die episodisch sind und deshalb nicht systematisch modelliert werden sollten. Nachfolgend bezeichnen $\lambda_0^i > 0$ und $\lambda_1^i = 1 - \lambda_0^i > 0$ die Wahrscheinlichkeiten, mit der Spieler i sich im Fehlerfall für Aktion 0 bzw. Aktion 1 entscheidet. Es ist wichtig anzunehmen, daß die Wahrscheinlichkeiten ε, λ_0^i und λ_1^i alle echt positiv sind.

Die Wahrscheinlichkeit $\bar{p}_i^\varepsilon(a_i^{t+1}|a^t)$, mit der Spieler i im Zustand a^t für die Folgeperiode die Aktion a_i^{t+1} wählt, ist nun gegeben durch

$$\bar{p}_i^\varepsilon(1|a^t) = (1-\varepsilon)\sum_{j=1}^{m} \mu_j^i d_j(a^t, i) + \varepsilon\lambda_1^i \tag{3.6}$$

und

$$\bar{p}_i^\varepsilon(0|a^t) = (1-\varepsilon)\sum_{j=1}^{m} \mu_j^i (1 - d_j(a^t, i)) + \varepsilon\lambda_0^i. \tag{3.7}$$

Die Übergangswahrscheinlichkeit von Zustand a zu Zustand a' beträgt dementsprechend

$$p_{aa'}^\varepsilon = \prod_{i=1}^{n} \bar{p}_i^\varepsilon(a_i'|a). \tag{3.8}$$

Der resultierende Markow-Prozeß (A, P^ε) wird *perturbiert* genannt. Er hat, wie auch im Beweis zu Lemma 3.1 gezeigt wird, die erfreuliche Eigenschaft, *reduzibel* zu sein, d. h. er hat genau eine Rekurrenzklasse, die den gesamten

Zustandsraum umfaßt. Die wichtigste Implikation dieser Eigenschaft besteht darin, daß die relative Häufigkeitsverteilung $(1/t) \cdot \sum_{\tau=0}^{t} a^0 \cdot (P^\varepsilon)^\tau$, die angibt, mit welcher Wahrscheinlichkeit sich der Markow-Prozeß für den Startzustand a^0 während der ersten t Perioden durchschnittlich in jedem Zustand befindet, für $t \to \infty$ gegen eine Verteilung h^ε konvergiert, die unabhängig von a^0 ist. Wie in der Literatur üblich, wird dann die ebenfalls von a^0 unabhängige Grenzverteilung $h^0 = \lim_{\varepsilon \to 0} h^\varepsilon$ betrachtet, damit das stochastische Rauschen nicht die Haupttriebkraft hinter den langfristigen Vorhersagen wird. Ein Zustand heißt nun nach Young (1993) *stochastisch stabil*, wenn h^0 ihn mit einer echt positiven Wahrscheinlichkeit belegt; stochastisch instabile Zustände sind demnach langfristig bedeutungslos. Es kann gezeigt werden, daß jeder stochastisch stabile Zustand in einer Rekurrenzklasse von (A, P) enthalten ist, während nicht alle rekurrenten Zustände auch stochastisch stabil sind. Die Einführung stochastischen Rauschens ermöglicht also deutlich schärfere Aussagen über die langfristigen Eigenschaften des Markow-Prozesses: Es existieren weniger "stabile" Zustände, die zudem noch unabhängig vom Startzustand sind. Zudem scheint die Annahme von Fehlern die Realitätsnähe des Modells zu erhöhen.

Leider ist die Berechnung der Grenzverteilung h^0 oft sehr schwierig. Es ist aber möglich, die stochastisch stabilen Zustände auch ohne Kenntnis dieser Verteilung zu identifizieren, wenn (A, P^ε) ein sogenannter *regulär perturbierter* Markow-Prozeß von (A, P) ist. Für das vorliegende Modell ist dieses tatsächlich der Fall:

Lemma 3.1 *(A, P^ε) ist ein regulär perturbierter Markow-Prozeß von (A, P).*

Young (1998, S. 55) hat dieselbe Aussage für ein Modell mit vergleichbar konstruiertem stochastischen Rauschen bewiesen. Der nachfolgende Beweis ist dementsprechend eine geradlinige Übertragung seiner Argumentation:

Beweis Zunächst ist eine zusätzliche Definition erforderlich: Bezeichne $E_{aa'} = \{i : \bar{p}_i(a'_i|a) = 0\}$ die Menge aller Spieler, die bei einem direkten Zustandsübergang von a nach a' einen Fehler machen müssen. Die Anzahl der für die-

sen Übergang erforderlichen Fehler $r_{aa'} = |E_{aa'}|$ heißt dann *Widerstand* dieses Übergangs.

Der Beweis besteht aus drei Teilen, die mit den drei definitionsgemäßen Eigenschaften regulär perturbierter Markow-Prozesse korrespondieren:

1. Annahmegemäß sind λ_0^i, $\lambda_1^i 0$ und ε echt positiv. Folglich sind auch $\bar{p}_i^\varepsilon(0|a)$ und $\bar{p}_i^\varepsilon(1|a)$ und letztlich auch die Übergangswahrscheinlichkeiten $p_{aa'}^\varepsilon$ echt positiv. Beim perturbierten Markow-Prozeß (A, P^ε) ist also jeder Zustand a' von jedem anderen Zustand a in nur einer Periode erreichbar; der Prozeß ist wie gefordert irreduzibel.

2. Qua Konstruktion gilt $\lim_{\varepsilon\to 0} \bar{p}_i^\varepsilon(a_i'|a) = \bar{p}_i(a_i'|a)$. Dann folgt wie gefordert

$$\lim_{\varepsilon\to 0} p_{aa'}^\varepsilon = \lim_{\varepsilon\to 0} \prod_{i=1}^n \bar{p}_i^\varepsilon(a_i'|a) = \prod_{i=1}^n \bar{p}_i(a_i'|a) = p_{aa'}.$$

3. Aus $\bar{p}_i^\varepsilon(a_i'|a) = \varepsilon\lambda_{a_i'}^i$ für alle $i \in E_{aa'}$ folgt

$$p_{aa'}^\varepsilon = \varepsilon^{r_{aa'}} \left(\prod_{i\in E_{aa'}} \lambda_{a_i'}^i \right) \left(\prod_{i\in I\setminus E_{aa'}} \bar{p}_i^\varepsilon(a_i'|a) \right).$$

   Dann ist

$$\lim_{\varepsilon\to 0} \frac{p_{aa'}^\varepsilon}{\varepsilon^{r_{aa'}}} = \left(\prod_{i\in E_{aa'}} \lambda_{a_i'}^i \right) \left(\prod_{i\in I\setminus E_{aa'}} \bar{p}_i(a_i'|a) \right).$$

   Dieser Grenzwert ist wie gefordert echt positiv, da $\lambda_{a_i'}^i > 0$ und für alle $i \in I \setminus E_{aa'}$ zudem $\bar{p}_i(a_i'|a) > 0$ gilt. ■

Nachfolgend werden nun die kurz- und langfristigen Eigenschaften dieses Modells unter drei verschiedenen Verhaltensannahmen untersucht.

3.3.3 Modellanalyse

3.3.3.1 Beste-Antwort-Dynamik

In diesem Abschnitt wird die Beste-Antwort-Regel analysiert. Diese Regel besagt, daß ein Spieler in der Erwartung, alle anderen würden ihre aktuelle Aktion

beibehalten, eine beste Antwort gibt. Formal kann sie für das betrachtete Spiel folgendermaßen formuliert werden:

$$\mathcal{B}(a,i) = \begin{cases} 1 & \sum_{j=1}^{n} a_j - a_i + 1 \geq \hat{n} \text{ and } s > 2r \\ 0 & \text{sonst} \end{cases} \tag{3.9}$$

Offensichtlich ist Aktion 1 nur dann eine beste Antwort, wenn die für die Folgeperiode erwartete Größe der Entwicklergemeinschaft, $\sum_{j=1}^{n} a_j - a_i + 1$, die kritische Schranke erreicht und der Gemeinschaftsnutzen s den durch einen Verkauf des Moduls erzielbaren Umsatz von $2r$ übersteigt.[28]

Des weiteren wird angenommen, daß die Spieler träge sind und nicht in jeder Periode ihre Aktion überdenken. Die Zahl der Entscheidungsregeln sei also $m = 2$ und die verwendeten Regeln $d_1 = \mathcal{B}$ und $d_2 = \mathcal{T}$. Dabei sei angemerkt, daß sowohl die Nachbarschaftsstruktur als auch die Trägheitsregel keinen Einfluß auf die kurz- und langfristigen Eigenschaften der Beste-Antwort-Dynamik haben. Hier werden sie lediglich zur Wahrung der Konsistenz mit den später betrachteten Dynamiken berücksichtigt.

Bei der kurzfristigen Analyse dieser Dynamik sind zwei Fälle zu unterscheiden. Ist der Gemeinschaftsnutzen klein ($s \leq 2r$), dann wird sich proprietäre Software vollständig durchsetzen; aufgrund der Beste-Antwort-Regel erkennen alle Spieler schließlich, daß die Kooperation in einer offenen Entwicklergemeinschaft eine dominierte Strategie ist. Ist der Gemeinschaftsnutzen hingegen so groß ($s > 2r$), daß umfassende Kooperation auch individuell rational ist, dann wird fast immer eines von zwei Regimen herrschen: Das bereits beschriebene proprietäre Regime oder ein offenes, in dem die Entwicklergemeinschaft alle Spieler umfaßt. Der folgende Satz präzisiert diese Aussage:

Satz 3.1 *Die unperturbierte Beste-Antwort-Dynamik hat immer den absorbierenden Zustand $z^0 = (0, \ldots, 0)$. Gilt $s > 2r$, dann ist zudem $z^1 = (1, \ldots, 1)$ absorbierend. Alle anderen Zustände sind transient.*

[28] Der Einfachheit halber wird angenommen, ein indifferenter Spieler wähle Aktion 0.

Beweis Im Zustand z^0 kann Aktion 1 niemals eine beste Antwort sein, da annahmegemäß $\hat{n} > 1$ gilt; folglich ist z^0 immer absorbierend. Gilt $s > 2r$, dann kann im Zustand z^1 überdies Aktion 0 niemals eine beste Antwort sein, da $\hat{n} \leq n$ ist; ergo ist dann auch z^1 absorbierend.

Alle anderen Zustände a mit $0 < g(a) < n$ sind hingegen transient: Gilt $s \leq 2r$ oder $g(a) < \hat{n} - 1$, dann ist für alle Spieler i die beste Antwort $\mathcal{B}(a, i) = 0$. Da nun die Wahrscheinlichkeit, daß alle Spieler die Beste-Antwort-Regel tatsächlich benutzen, echt positiv ist, kann der Markow-Prozeß in nur einem Schritt von a in den absorbierenden Zustand z^0 übergehen. Gilt hingegen $s > 2r$ und $g(a) \geq \hat{n}-1$, dann ist für alle Spieler i mit $a_i = 0$ die beste Antwort $\mathcal{B}(a, i) = 1$. Da auch die Wahrscheinlichkeit, daß alle Spieler mit $a_i = 0$ die Beste-Antwort-Regel und alle Spieler mit $a_i = 1$ die Trägheitsregel benutzen, echt positiv ist, kann der Markow-Prozeß in nur einem Schritt von a in den absorbierenden Zustand z^1 übergehen. ■

Offensichtlich replizert Satz 3.1 die Nash-Gleichgewichte des Stufenspiels in der Terminologie markowscher Prozesse. Genau deshalb kann im folgenden für den relevanten Fall $s > 2r$ das Konzept der stochastischen Stabilität als Kriterium für die Selektion von Gleichgewichten dienen:

Satz 3.2 *Die stochastisch stabilen Zustände der perturbierten Beste-Antwort-Dynamik sind*

- *für $s \leq 2r$ nur der Zustand z^0 und*
- *für $s > 2r$ der Zustand z^0, genau dann wenn $\hat{n} \geq n/2 + 1$ gilt, und der Zustand z^1, genau dann wenn $\hat{n} \leq n/2 + 1$ gilt.*

Beweis Für $s \leq 2r$ ist z^0 laut Satz 3.1 der einzige in einer Rekurrenzklasse der unperturbierten Beste-Antwort-Dynamik enthaltene Zustand; folglich ist er auch der einzige stochastisch stabile Zustand.

Für $s > 2r$ sind hingegen nach Satz 3.1 z^0 und z^1 die einzigen möglichen stochastisch stabilen Zustände. Lemma 3.1 ermöglicht nun die Anwendung ei-

nes zentralen Resultats Youngs (1998, S. 56), nach dem stochastisch stabile Zustände dadurch charakterisiert sind, daß sie Elemente von Rekurrenzklassen mit minimalem stochastischen Potential sind. Für den hier vorliegenden Fall zweier Rekurrenzklassen $\{z^0\}$ und $\{z^1\}$ entspricht das stochastische Potential einer Klasse dabei der minimalen Anzahl von Fehlern, die erforderlich sind, um sie von der anderen Klasse kommend zu erreichen.[29]

Die minimale Anzahl von Fehlern, die von $\{z^0\}$ in $\{z^1\}$ führt, beträgt offensichtlich $\hat{n} - 1$. Erst dann wird Aktion 1 für alle verbleibenden Spieler i mit $a_i = 0$ eine beste Antwort. Das stochastische Potential von $\{z^1\}$ ist also $\rho_{z^1} = \hat{n} - 1$. In der Gegenrichtung sind hingegen $n - \hat{n} + 1$ Fehler erforderlich, damit Aktion 0 für alle Spieler i mit $a_i = 1$ eine beste Antwort wird. Das stochastische Potential von $\{z^0\}$ ist also $\rho_{z^0} = n - \hat{n} + 1$. Ein einfacher Vergleich von ρ_{z^0} und ρ_{z^1} erbringt nun den zweiten Teil der Satzaussage. ■

Satz 3.2 postuliert, die *langfristige* Existenz offener Entwicklergemeinschaften sei an zwei Voraussetzungen geknüpft. Erstens müßten sie wie bereits bei der kurzfristigen Betrachtung individuell rational sein ($s > 2r$). Zudem dürfe aber auch das zu lösende Koordinationsproblem nicht zu schwierig sein. Während die erste Voraussetzung offensichtlich ist, verdient die zweite eine eingehendere Diskussion.

Die im Satz genannte Bedingung für die stochastische Stabilität von z^1, $\hat{n} \leq n/2+1$, entspricht wegen $\hat{n} = \lfloor \zeta n \rfloor$ für großes n näherungsweise der einfacheren Bedingung $\zeta < 1/2$. D. h. ein großes, aus vielen Modulen bestehendes Softwaresystem wird langfristig nur unter einer offenen Lizenz angeboten, wenn es bereits in halbfertigem Zustand nützlich ist. In der Realität wird diese Voraussetzung aber so gut wie nie erfüllt sein. Beispielsweise ist für ein Unternehmen, welches die Software wie bei Henkel (2004b) in eigenen Produkten verwenden will, ein solches Stückwerk quasi wertlos. Bezüglich der langfristigen Verfügbarkeit offener Software stimmt Satz 3.2 also eher pessimistisch.

[29] Die allgemeine Definition des stochastischen Potentials beinhalten mehrere komplizierte graphentheoretische Konzepte. Sie findet sich etwa bei Young (1998, S. 54 ff).

Nachfolgend wird allerdings gezeigt, daß es nicht robust gegenüber einer naheliegenden, minimalen Veränderung der Verhaltensannahmen ist. Denn führt man wie bei Eshel et al. (1998) zusätzlich imitationales Verhalten ein, ändern sich die langfristigen Eigenschaften des perturbierten Markow-Prozesses drastisch. Dabei wird zunächst eine reine Imitationsdynamik betrachtet, um die so entstehenen Effekte in Reinkultur analysieren zu können. Abschließend werden dann die Imitations- und die Beste-Antwort-Dynamik zu einer hybriden Dynamik verschmolzen.

3.3.3.2 Imitationsdynamik

Die Berücksichtigung imitationalen Verhaltens ist aus mehreren Gründen angebracht: Zunächst einmal handelt es sich schlicht um eine naheliegende Verhaltensweise, wenn der Erfolg anderer sichtbar ist; faktisch ist Imitation sogar diejenige Verhaltensregel, die alle Menschen nach ihrer Geburt als erste verwenden. Zudem belegen diverse Laborexperimente wie etwa die von Pingle und Day (1996) sowie Duffy und Feltovich (1999) die wichtige Rolle imitationalen Verhaltens auch in ökonomischen Entscheidungssituationen.[30]

Im Modell betrachtet jeder Spieler i wie bei Eshel et al. (1998) nur seine Nachbarschaft $N(i)$, da es für ihn schwierig sein sollte, das Nutzenniveau solcher Spieler, mit denen er nicht interagiert, zu bestimmen. Dabei vergleicht der Spieler die mit den beiden Aktionen 0 und 1 jeweils maximal erzielten Nutzenniveaus $\pi_0(a,i) = \max_{j\in N(i)} u_j(a)(1-a_j)$ und $\pi_1(a,i) = \max_{j\in N(i)} u_j(a)a_j$ und wählt seine Aktion für die Folgeperiode entsprechend.[31] Formal lautet diese

[30] Eine Übersicht über die Literatur und eine mikroökonomische Herleitung imitationalen Verhaltens gibt Schlag (1998).

[31] Bei Eshel et al. (1998) vergleichen die Spieler die *durchschnittlichen* Nutzenniveaus. In dieser Arbeit werden hingegen maximale Nutzenniveaus betrachtet, da dies die Analyse für Nachbarschaften beliebiger Größe stark vereinfacht. Eshel et al. (1998) analysieren ihr Modell nur für die Radien $r = 1$ und $r = 2$.

Regel

$$\mathcal{I}(a,i) = \begin{cases} 1 & \pi_1(a,i) > \pi_0(a,i) \\ 0 & \text{sonst} \end{cases} . \tag{3.10}$$

Abweichend von Eshel et al. (1998) wird den Spielern wiederum Trägheit unterstellt. Zum einen erscheint diese Annahme realitätsnah – warum sollten alle Spieler ihre Aktionen stets im Gleichschritt anpassen? Zum anderen verhindert sie auch permanentes Koordinationsversagen, welches zu einer zyklischen Entwicklung des unperturbierten Markow-Prozesses führen kann und so die langfristige Analyse kompliziert. Die Zahl der Entscheidungsregeln sei also $m = 2$ und die verwendeten Regeln $d_1 = \mathcal{I}$ und $d_2 = \mathcal{T}$.

Für die kurzfristige Analyse erweist sich das nachfolgend angegebene Lemma als hilfreich. Es besagt, daß in solchen absorbierenden Zuständen, die beide möglichen Aktionen enthalten, die kooperative Aktion 1 nur in hinreichend großen, zusammenhängenden Gruppen auftreten kann. Derartige Gruppen von Spielern, die alle dieselbe Aktion verwenden, werden als *1-Ketten* bzw. *0-Ketten* bezeichnet; des weiteren heiße ein Zustand $a \notin \{z^0, z^1\}$ *gemischt*.

Lemma 3.2 *In jedem gemischten absorbierenden Zustand der unperturbierten Imitationsdynamik haben alle 1-Ketten mindestens die Länge* $2r + 1$.

Beweis Zunächst wird gezeigt, daß kein gemischter absorbierender Zustand eine 1-Kette mit einer Länge $l \in \{r + 1, \ldots, 2r\}$ enthalten kann. Anschließend wird dieselbe Aussage für 1-Ketten mit einer Länge $l \in \{1, \ldots, r\}$ bewiesen.[32]

1. Sei $a \in A$ ein gemischter Zustand mit einer 1-Kette der Länge $l \in \{r + 1, \ldots, 2r\}$. Ein Teil dieses Zustands sieht dann folgendermaßen aus:[33]

[32] Alle Beweise zur Imitationsdynamik verwenden bestimmte geometrische Anordnungen von Spielern auf dem Kreis, die nur möglich sind, wenn n ein bestimmtes, geringes Vielfaches von r übersteigt; im Fall dieses Lemmas muß beispielsweise $n > 3r$ gelten. Da aber die Einführung einer Nachbarschaftstruktur die Mühe überhaupt nur dann wirklich lohnt, wenn n deutlich größer als r ist, wird dieser Aspekt im folgenden aus Gründen der Einfachheit vernachlässigt.

[33] Diese Darstellung zeigt wie alle noch folgenden nur den relevanten Teil des Zustands als

$$\overset{\alpha-r}{\ldots\ a_{\alpha-r}}\ \ldots\quad \overset{\alpha}{0}\quad \underbrace{\overset{\beta}{1}\quad\ldots\quad \overset{\gamma-r}{1}\quad\ldots\quad \overset{\beta+r}{1}\quad\ldots\quad \overset{\gamma}{1}}_{l}\quad \overset{\delta}{0}\quad\ldots$$

Sei ohne Beschränkung der Allgemeinheit $u_{\gamma-r}(a) \geq u_{\beta+r}(a)$. Die beiden benachbarten Spieler α und β teilen sich die Nachbarschaft $\hat{N} = N(\alpha) \cap N(\beta)$. Wegen $\beta = \alpha + 1$, $\gamma = \beta + l - 1$ und $l \leq 2r$ ist auch Spieler $\gamma - r$ in $\hat{N}$ enthalten. Da nun $u_{\gamma-r}(a) \geq u_{\beta+r}(a)$ und $a_{\gamma-r} = a_{\beta+r} = 1$ gilt, wird Spieler β nicht den Spieler $\beta + r$ imitieren, sondern einen Spieler aus $\hat{N}$.[34]

Imitierte auch α einen Spieler aus $\hat{N}$, so würde entweder er oder β seine Aktion ändern. Der Zustand a kann also nur dann absorbierend sein, wenn Spieler $\alpha - r$ Aktion 0 spielt und von α imitiert wird. Gerade das ist aber nicht möglich, wie der folgende Nutzenvergleich zeigt:

$$\begin{aligned}
u_{\alpha-r}(a) &= 2r + (1+v)\left(\sum_{i=\alpha-2r}^{\alpha-r-1} a_i + \sum_{i=\alpha-r+1}^{\alpha} a_i\right) \\
&\leq 2r + (1+v)\left(r + \sum_{i=\alpha-r+1}^{\alpha} a_i\right) \\
&= 2r + (1+v)\left(r + \sum_{i=\alpha-r}^{\alpha-1} a_i\right) \\
&= u_\alpha(a)
\end{aligned}$$

Folglich kann es mit einer echt positiven Wahrscheinlichkeit geschehen, daß α und β beide einen Spieler aus $\hat{N}$ imitieren und in der nächsten Periode dieselbe Aktion verwenden. Zustand a ist also nicht absorbierend.

2. Sei nun $a \in A$ ein gemischter Zustand mit einer 1-Kette der Länge $l \in \{1, \ldots, r\}$. Ein Teil dieses Zustands hat also die folgende Gestalt:

$$\ldots\quad \overset{\alpha}{0}\quad \underbrace{\overset{\beta}{1}\quad 1\quad \ldots\quad 1\quad 1}_{l}\quad 0\quad \ldots$$

gerade Linie. Die griechischen Kleinbuchstaben über den Aktionen bezeichnen die Indizes der Spieler; die geschweifte Klammer markiert eine Kette der Länge l.

[34] Die Beschreibung des Beweises wird leichter, wenn man ohne Beschränkung der Allgemeinheit unterstellt, die Spieler α und β würden bei gleichen Nutzenniveaus und Aktionen einen Spieler aus $\hat{N}$ imitieren.

Wiederum kann Zustand a nur dann absorbierend sein, wenn Spieler $\alpha - r$ Aktion 0 spielt und von α imitiert wird oder Spieler $\beta + r$ Aktion 1 spielt und von β imitiert wird. Wie sich allerdings zeigt, kann keine der beiden Bedingungen je erfüllt sein.

Zunnächst wird der Fall betrachtet, daß Spieler $\alpha - r$ Aktion 0 spielt. Verwenden nun alle Spieler zwischen $\alpha - r$ und α ebenfalls Aktion 0, dann gilt $u_{\alpha-r+1}(a) = u_{\alpha-r}(a) + (1+v)\,(a_\beta - a_{\alpha-2r}) \geq u_{\alpha-r}(a)$; Spieler α wird also wegen $\alpha - r + 1 \in \hat{N}$ nicht $\alpha - r$ imitieren. Verwendet hingegen ein Spieler zwischen $\alpha - r$ und α, nachfolgend mit γ bezeichnet, die Aktion 1, dann sieht ein Teil von Zustand a wie folgt aus:

$$\begin{array}{cccccccccc} & & \overset{\alpha-r}{} & & \overset{\gamma}{} & & \overset{\alpha}{} & \overset{\beta}{} & & \\ \ldots & & 0 & \ldots & 1 & \ldots & 0 & 1 & 1 & \ldots \end{array}$$

Imitiert nun α den Spieler $\alpha - r$, dann kann γ seine derzeitige Aktion langfristig nur dann beibehalten, wenn ein Spieler δ links von $\alpha - r$ mit $a_\delta = 1$ und $u_\delta(a) > u_{\alpha-r}(a)$ existiert. Wenn aber ein solcher Spieler δ existiert, dann kann $\alpha - r$ seine derzeitige Aktion langfristig nur dann beibehalten, wenn ein Spieler ε links von δ mit $a_\varepsilon = 0$ und $u_\varepsilon(a) \geq u_\delta(a)$ existiert. Die folgende Darstellung veranschaulicht dies:

$$\begin{array}{cccccccccccc} & \overset{\varepsilon}{} & & \overset{\delta}{} & & \overset{\alpha-r}{} & & \overset{\gamma}{} & & \overset{\alpha}{} & \overset{\beta}{} & \\ \ldots & 0 & \ldots & 1 & \ldots & 0 & \ldots & 1 & \ldots & 0 & 1 & 1 & \ldots \end{array}$$

Offensichtlich kann diese Kette von Stabilitätsbedingungen mit stetig steigenden Anforderungen an die Nutzenniveaus aufgrund der zirkulären Anordnung der Spieler nicht ad infinitum fortgesetzt werden.

Eine analoge Argumentation erbringt, daß a auch dann nicht absorbierend sein kann, wenn Spieler $\beta + r$ Aktion 1 spielt und von β imitiert wird. ■

Lemma 3.2 ermöglicht es, nun die Rekurrenzklassen der unperturbierten Imitationsdynamik zu bestimmen. Es wird sich zeigen, daß alle Rekurrenzklassen aus jeweils einem absorbierenden Zustand bestehen. Zwei offensichtliche absorbierende Zustände sind z^0 und z^1. Abhängig vom Wert des Quelltexts v und

dem Gemeinschaftsnutzen s existieren mitunter aber noch zahlreiche gemischte absorbierende Zustände: Deren wichtigste Eigenschaften sind in Abbildung 3.3 dargestellt;[35] die Abbildung erschließt sich dabei am besten anhand zweier Lesebeispiele:

- Sei $(v, s) = (v_1, s_1)$ wie in der Abbildung angegeben. Dann ist ein gemischter Zustand a mit $g(a) \geq \hat{n}$ genau dann absorbierend, wenn alle in ihm enthaltenen 0-Ketten genau die Länge 2 und alle 1-Ketten wie bereits bekannt mindestens die Länge $2r + 1$ aufweisen. Des weiteren ist ein gemischter Zustand a mit $g(a) < \hat{n}$ genau dann absorbierend, wenn alle in ihm enthaltenen 0-Ketten genau die Länge $r - 1$ und alle 1-Ketten wiederum mindestens die Länge $2r + 1$ aufweisen. Man beachte, daß der Fall $g(a) < \hat{n}$ formal mit dem Fall $g(a) \geq \hat{n}$ und $s = 0$ identisch ist, dementsprechend erhält man dann die Länge der 0-Ketten, indem man vom Punkt (v_1, s_1) das Lot auf die v-Achse fällt; die stabile Länge beträgt $r - 1$.

- Sei nun $(v, s) = (v_2, s_2)$. Dann existiert kein gemischter absorbierender Zustand a mit $g(a) \geq \hat{n}$. Nur Zustände a mit $g(a) < \hat{n}$ sind absorbierend, und zwar genau dann, wenn alle 0-Ketten mindestens die Länge $r+1$ und alle 1-Ketten wiederum mindestens die Länge $2r + 1$ aufweisen.

Der folgende Satz präzisiert diese Ausführungen:

Satz 3.3 *Die unperturbierte Imitationsdynamik hat immer die absorbierenden Zustände z^0 und z^1. Zusätzlich sind noch alle Zustände a absorbierend, deren 1-Ketten mindestens die Länge $2r + 1$ aufweisen und die eine der folgenden Bedingungen erfüllen:*[36]

[35] Man beachte, daß Abbildung 3.3 in der gegebenen Form nur für $r \geq 5$ gültig ist. Um sie an kleinere Werte von r anzupassen, müßte man einige der diagonal verlaufenden Geraden entfernen.

[36] Für $r = 1$ sollte der in den Bedingungen 1 und 2 auftretende Term $(r + 1)/(r - 1)$ als ∞ interpretiert werden; Bedingung 1 kann also niemals erfüllt sein, während sich die in Bedingung 2 formulierte Anforderung an v zu $1 < v$ vereinfacht.

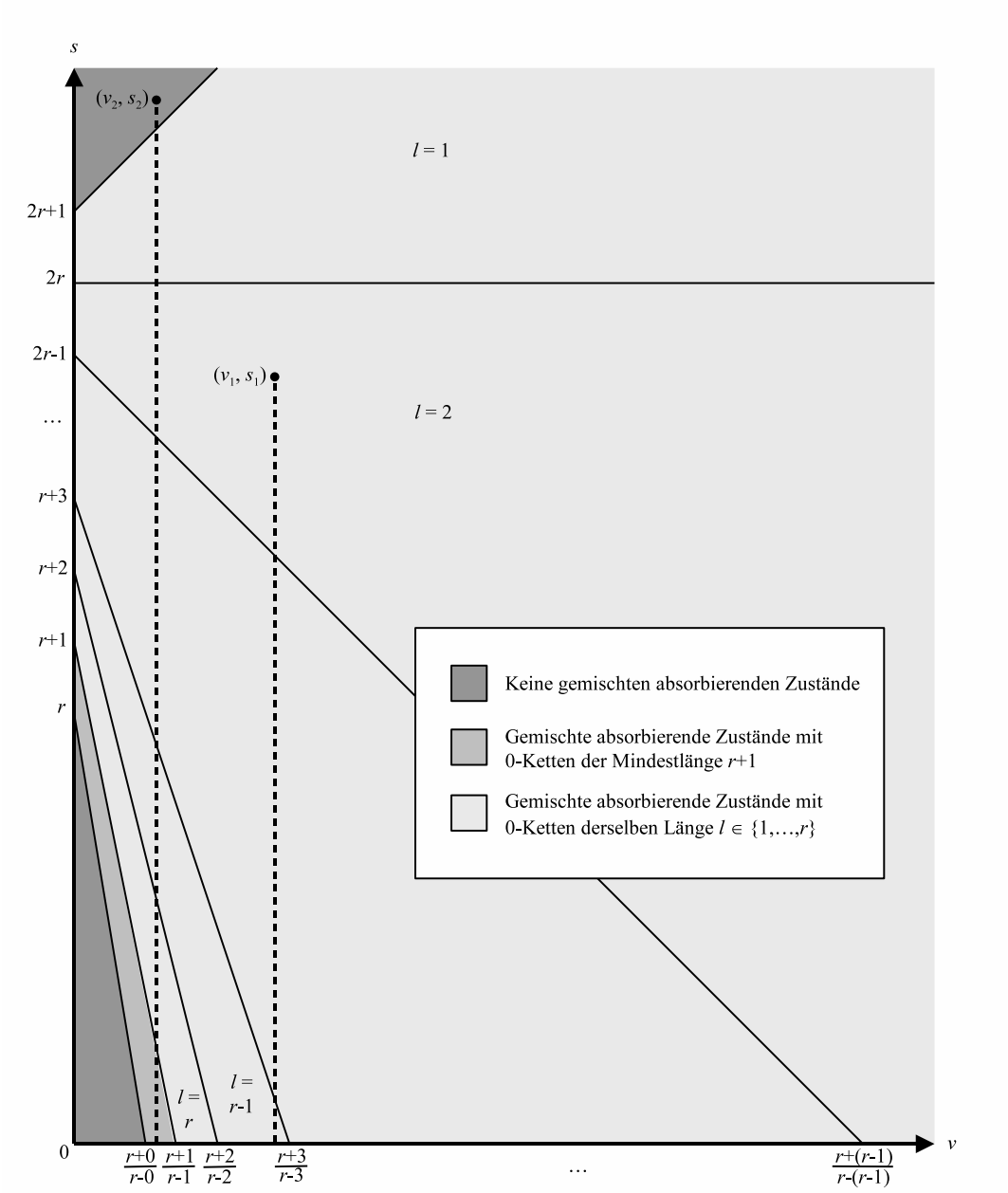


Abbildung 3.3: Gemischte absorbierende Zustände der unperturbierten Imitationsdynamik mit $g(a) \geq \hat{n}$

1. *$g(a) < \hat{n}$ und $(r+1)/(r-1) < v$ und alle 0-Ketten haben genau die Länge $\hat{l}_0(r,v) = \lfloor 2r/(1+v) + 2 \rfloor \in \{2, \ldots, r\}$*

2. *$g(a) < \hat{n}$ und $1 < v \leq (r+1)/(r-1)$ und alle 0-Ketten haben mindestens die Länge $r+1$*

3. *$g(a) \geq \hat{n}$ und $r + 1 - (r-1)v < s \leq 2r + 1 + v$ und alle 0-Ketten haben genau die Länge $\hat{l}_1(r,v,s) = \lfloor (2r-s)/(1+v) + 2 \rfloor \in \{1, \ldots, r\}$*

4. *$g(a) \geq \hat{n}$ und $r - rv < s \leq r + 1 - (r-1)v$ und alle 0-Ketten haben mindestens die Länge $r+1$*

Alle anderen Zustände sind transient.

Beweis Die Zustände z^0 und z^1 sind für rein imitationales Verhalten trivialerweise absorbierend. Des weiteren ist aus Lemma 3.2 bereits die erforderliche Mindestlänge der in gemischten absorbierenden Zuständen enthaltenen 1-Ketten bekannt. Was verbleibt, ist damit die Berechnung der Längen stabiler 0-Ketten:

Dazu betrachte man einen gemischten Zustand a mit 1-Ketten der erforderlichen Länge und einer 0-Kette der Länge $l \geq 1$. Ausschnittsweise hat a also die folgende Gestalt:

$$\ldots \quad \overset{\alpha}{1} \quad \underbrace{\overset{\beta}{1} \quad \ldots \quad \overset{\gamma}{1}}_{r} \quad \underbrace{\overset{\delta}{0} \quad \ldots \quad 0}_{l} \quad 1 \quad \ldots \quad 1 \quad 1 \quad \ldots$$

Zunächst wird der Fall $g(a) \geq \hat{n}$ betrachtet. Da die 1-Ketten mindestens die Länge $2r+1$ haben, gilt dann $u_\alpha(a) = 2r(1+v)+s$, $u_\beta(a) = (2r-1)(1+v)+s$ und $u_\delta(a) = (2r+1-\min\{l, r+1\})(1+v) + 2r$.

Die 0-Kette ist genau dann stabil, wenn Spieler γ und Spieler δ bei der Verwendung der Imitationsregel ihre jeweilige Aktion beide beibehalten.[37] Spieler

[37] Wenn die Spieler am Rand einer Kette ihre Aktion beibehalten, dann wechseln Spieler in ihrem Inneren offensichtlich ebenfalls nicht.

γ wechselt genau dann nicht, wenn $\pi_1(a,\gamma) = u_\alpha(a) > \pi_0(a,\gamma) = u_\delta(a)$ gilt; diese Bedingung vereinfacht sich zu

$$\min\{l, r+1\} > \frac{2r-s}{1+v} + 1. \tag{3.11}$$

Analog wechselt Spieler δ genau dann nicht, wenn $\pi_1(a,\delta) = u_\beta(a) \leq \pi_0(a,\delta) = u_\delta(a)$ gilt, oder vereinfacht

$$\min\{l, r+1\} \leq \frac{2r-s}{1+v} + 2. \tag{3.12}$$

Für $s > 2r+1+v$ hat Gleichung (3.12) keine Lösung. Spieler δ wechselt immer und selbst 0-Ketten der Länge 1 sind nicht stabil. In Abbildung 3.3 entspricht dies der dunkelgrauen Fläche oben links. Für $r+1-(r-1)v < s \leq 2r+1+v$ haben die Gleichungen (3.11) und (3.12) genau eine Lösung, nämlich $\hat{l}_1(r,v,s) = \lfloor (2r-s)/(1+v)+2 \rfloor$. In Abbildung 3.3 entspricht dies der hellgrauen Fläche. Für $r - rv < s \leq r+1-(r-1)v$ lösen hingegen alle $l \geq r+1$ die beiden Gleichungen. In Abbildung 3.3 entspricht dies der mittelgrauen Fläche. Schließlich hat für $s \leq r - rv$ Gleichung (3.11) keine Lösung. Spieler γ wechselt immer und die 1-Ketten sind nicht stabil. In Abbildung 3.3 entspricht dies der dunkelgrauen Fläche unten links.

Damit sind die im Satz unter 3 und 4 aufgeführten Bedingungen hergeleitet. Für die Punkte 1 und 2 wiederhole man die obige Argumentation mit $s = 0$, was äquivalent zu $g(a) < \hat{n}$ ist.

Damit sind alle absorbierenden Zustände bekannt. Die Tatsache, daß keine mehrelementigen Rekurrenzklassen existieren, folgt trivialerweise aus der Annahme trägen Verhaltens.[38] ■

Unter der Imitationsdynamik kann also die Beteiligung an einer offenen Entwicklergemeinschaft kurzfristig stabil sein, selbst wenn dies nicht individuell rational ist. Zudem ist dieses Verhalten in den gemischten absorbierenden Zuständen für die meisten Werte von v und s überraschend populär. Denn im gesamten hellgrauen Bereich von Abbildung 3.3 haben alle 1-Ketten mindestens

[38] Eshel et al. (1998) diskutieren die Wirkung von Trägheit ausführlicher.

die Länge $2r + 1$ und alle 0-Ketten höchstens die Länge r. Da aufgrund der Kreisstruktur die Anzahl der 1-Ketten der der 0-Ketten entspricht, engagieren sich dann über 2/3 der Gesamtpopulation in der Entwicklergemeinschaft.

Man könnte nun vermuten, zumindest für einen geringen Gemeinschaftsnutzen s wäre diese kurzfristige Popularität langfristig instabil. Denn ein Spieler, der inmitten einer 1-Kette aufgrund eines Fehlers zur Aktion 0 wechselt, erhält einen sehr hohen Nutzen und wird daher imitiert, während ein Spieler, der inmitten einer 0-Kette wegen eines Fehlers zur Aktion 1 wechselt, aufgrund seines niedrigen Nutzens nicht imitiert werden wird. Fehler scheinen sich also zugunsten von Aktion 0 auszuwirken. Der folgende Satz widerlegt aber diese Intuition für einen Teil des Parameterraums sogar für $s = 0$:

Satz 3.4 *Gilt $v > 2r - 1$, $r > 1$ und $s = 0$, dann ist z^0 für hinreichend großes n kein stochastisch stabiler Zustand der perturbierten Imitationsdynamik.*

Beweis Der Beweis erfordert ein wenig zusätzliche Notation: Offensichtlich ist für jedes n die Anzahl der 0-Ketten, die in einem absorbierenden Zustand auftreten kann, nach oben beschränkt; sei nachfolgend $w(n)$ diese Schranke. Für alle $k \in \{1, \ldots, w(n)\}$ bezeichne zudem $Z(k)$ die Menge aller gemischten absorbierenden Zustände mit genau k 1- und 0-Ketten. Schließlich sei $Z = \bigcup_{k=1}^{w(n)} Z(k)$ die Menge aller gemischten absorbierenden Zustände. Dem eigentlichen Beweis gehen die dreier Hilfssätze voran:

Hilfssatz 1 *Mit nur einem Fehler kann ausgehend von z^1 ein Zustand aus der Menge $Z(1)$ erreicht werden:* Wenn im Zustand z^1 ein Spieler aufgrund eines Fehlers zur Aktion 0 wechselt, dann erhält er wegen $s = 0$ den höchstmöglichen Nutzen. Es ist dann möglich, daß ihn ein unmittelbarer Nachbar imitiert, während alle anderen Spieler träge sind. Das Ergebnis ist eine 1-Kette der Länge 2, die wegen $v > 2r - 1$, $r > 1$ und $s = 0$ wie aus Abbildung 3.3 ersichtlich stabil ist. Folglich wurde mit nur einem Fehler ein Zustand aus $Z(1)$ erreicht. □

Hilfssatz 2 *Die kleinste Anzahl an Fehlern, mit der ausgehend von z^0 ein Zustand aus der Menge $Z(1)$ erreicht werden kann, beträgt $r + 2$:* Offensichtlich

werden Spieler, die fälschlicherweise von Aktion 0 zu Aktion 1 wechseln, am ehesten imitiert, wenn sie eine 1-Kette bilden. Man nehme also an, im Zustand z^0 wechselten $l \in \mathbb{N}$ unmittelbar benachbarte Spieler aufgrund von Fehlern zur Aktion 1. Das Ergebnis ist ein Zustand a der folgenden Gestalt:

$$\ldots \quad 0 \quad \ldots \quad \overset{\alpha}{0} \quad \underbrace{1 \quad \ldots \quad 1 \quad \ldots \quad 1}_{l} \quad 0 \quad \ldots \quad 0 \quad \ldots$$

Die 1-Kette kann genau dann wachsen, wenn Spieler α Aktion 1 imitiert. Dieses ist genau dann der Fall, wenn

$$\pi_1(a, \alpha) = \min\{l - 1, 2r - 1\}(1 + v) > \pi_0(a, \alpha) = \min\{l, r\}(1 + v) + 2r \quad (3.13)$$

gilt. Es ist nun leicht zu überprüfen, daß für $v > 2r - 1$, $r > 1$ und $s = 0$ der kleinste Wert für l, der Ungleichung (3.13) erfüllt, $r + 2$ ist. Wenn nun aber eine 1-Kette der Länge $r + 2$ wachsen kann, so kann es eine längere 1-Kette erst recht. Dieses Wachstum kann nun durch die Imitationsregel allein so lange voranschreiten, bis die 0-Kette auf die stabile Länge 2 reduziert und somit ein Zustand aus $Z(1)$ erreicht wurde. □

Hilfssatz 3 *Für jedes Paar gemischter absorbierender Zustände $(z_0, \bar{z}) \in Z^2$ mit $z_0 \neq \bar{z}$ existiert eine endliche Folge von $w \in \mathbb{N}$ absorbierenden Zuständen $(z_1, \ldots, z_w) \in Z^w$ mit $z_w = \bar{z}$, so daß für alle $f \in \{1, \ldots, w\}$ genau ein Fehler erforderlich ist, um z_f ausgehend von z_{f-1} zu erreichen:* Nachfolgend wird ein Algorithmus dargestellt, mit dem die beschriebene Folge $(z_1, \ldots, z_w)$ in maximal drei Schritten konstruiert werden kann. Selbstverständlich bricht der Algorithmus sofort ab, sobald $\bar{z}$ erreicht wurde.

1. Der erste Schritt besteht in der Konstruktion des ersten Teils der Folge $(z_1, \ldots, z_{f'})$, so daß $z_{f'} \in Z(1)$ ist. Gilt bereits $z_0 \in Z(1)$, so setze man $f' = 0$ und gehe direkt zu Schritt 2. Anderenfalls setze man k_0 so, daß $z_0 \in Z(k_0)$ ist und fixiere eine in z^0 enthaltene 1-Kette. Ist die Länge dieser 1-Kette größer als $2r + 1$, so kann sie durch genau einen Fehler um 1 reduziert werden, indem der Spieler an ihrem rechten Rand fehlerhaft zur Aktion 0 wechselt. Dieses führt in einen transienten Zustand, da die

resultierende 0-Kette zu lang ist. Allerdings kann diese 0-Kette wieder auf die stabile Länge schrumpfen, indem der Spieler an ihrem rechten Rand mit der Imitationsregel zu Aktion 1 wechselt. Nach endlich vielen Wiederholungen dieses Vorgehens erhält man schließlich eine Folge absorbierender Zustände, die durch jeweils einen Fehler getrennt sind und z^0 mit einem absorbierenden Zustand verbinden, in dem die anfangs fixierte 1-Kette genau die Länge $2r + 1$ hat. Von diesem Zustand kann man wiederum mit nur einem Fehler des Spielers, der in der Mitte der fixierten 1-Kette angesiedelt ist, einen absorbierenden Zustand in $Z(k_0 - 1)$ erreichen. Denn wechselt dieser Spieler zur Aktion 0, so erhält er wegen $s = 0$ den größtmöglichen Nutzen. Folglich können ihn alle Spieler in seiner Nachbarschaft gleichzeitig imitieren, was die gesamte 1-Kette zerstört. Die resultierende 0-Kette ist natürlich viel zu lang, kann aber allein durch Imitation auf die stabile Länge schrumpfen, was in einem Zustand aus $Z(k_0 - 1)$ endet. Die Wiederholung dieses Vorgehens mündet schließlich in einen Zustand $z_{f'} \in Z(1)$.

2. Der nächste Schritt umfaßt die Konstruktion des zweiten Teils der Folge $(z_{f'+1}, \ldots, z_{f''})$, so daß $z_{f''} \in Z(1)$ *und* die Position der in $z_{f''}$ enthaltenen 0-Kette der einer in $\bar{z}$ enthaltenen entspricht. Sind die 0-Ketten bereits gleich positioniert, so setze man $f'' = f'$ und gehe direkt zu Schritt 3. Anderenfalls kann man die in $z_{f'}$ enthaltene 0-Kette mit nur einem Fehler um 1 nach rechts verschieben, in dem der Spieler an ihrem linken Rand aufgrund eines Fehlers zur Aktion 1 wechselt. Dieses führt in einen transienten Zustand, da die resultierende 0-Kette zu kurz ist. Allerdings kann sie wieder auf die stabile Länge wachsen, indem der Spieler am linken Rand der 1-Kette mit der Imitationsregel zu Aktion 0 wechselt. Der resultierende Zustand ist wieder absorbierend. Die Wiederholung dieses Vorgehens mündet schließlich über eine Folge von durch nur einen Fehler getrennten absorbierenden Zuständen in $z_{f''} \in Z(1)$.

3. Der dritte Schritt besteht in der Konstruktion des letzten Teils der Folge

$(z_{f''+1}, \ldots, z_w)$ mit $z_w = \bar{z}$. Hierfür müssen noch all jene 0-Ketten, die zwar in $\bar{z}$ aber nicht in $z_{f''}$ enthalten sind, hinzugefügt werden. Jede dieser 0-Ketten kann mit nur einem Fehler eingefügt werden, indem der Spieler am linken Rand einer nur in $\bar{z}$ enthaltenen 0-Kette aufgrund eines Fehlers zur Aktion 0 wechselt. Diese 0-Kette kann nun allein durch Imitation so lange nach rechts wachsen, bis sie die stabile Länge erreicht hat. Der resultierende Zustand ist dann wieder absorbierend. Die Wiederholung dieses Vorgehens führt schließlich zum Zustand $\bar{z}$. □

Aus den Hilfssätzen 1 bis 3 folgt unmittelbar, daß das stochastische Potential jedes gemischten absorbierenden Zustands nach oben durch $r+2+|Z|$ beschränkt ist.[39]

Bezeichnet $h(n)$ für alle n die minimale Anzahl von Fehlern, die erforderlich ist, um z^0 ausgehend von einem Zustand aus $Z \cup \{z^1\}$ zu erreichen, so ist das stochastische Potential von z^0 nach unten offensichtlich durch $|Z| + h(n)$ beschränkt. Da nun nach Young (1993) der Zustand z^0 nicht stochastisch stabil ist, wenn er nicht das kleinste stochastische Potential aufweist, genügt es im weiteren zu zeigen, daß für hinreichend großes n die Ungleichung $h(n) > r + 2$ gilt.

Offensichtlich muß in jeder 1-Kette eines Zustands aus Z mindestens ein Fehler auftreten, um in den Attraktionsbereich von z^0 zu gelangen. Wir können daher die weitere Betrachtung auf alle absorbierenden Zustände aus der Menge $Z' = \{z^1\} \cup \bigcup_{k=1}^{\min\{r+2,w(n)\}} Z(k)$ beschränken, da alle anderen gemischten absorbierenden Zustände mehr als $r + 2$ 1-Ketten enthalten und daher auch mehr als $r + 2$ Fehler erforderlich sind.

Da nun die Anzahl der in den Zuständen aus Z' enthaltenen 1-Ketten nach oben beschränkt ist, muß für $n \to \infty$ die Länge mindestens einer 1-Kette ebenfalls gegen ∞ divergieren. Da für $v > 2r - 1$, $r > 1$ und $s = 0$ nach Abbildung 3.3 die Länge der stabilen 0-Ketten 2 beträgt, kann ein Spieler, der inmitten

[39] Die Definition des stochastischen Potentials findet sich bei Young (1998, S. 54 ff).

einer 1-Kette aufgrund eines Fehlers zur Aktion 0 wechselt, höchstens $2r$ weitere Spieler konvertieren. Das impliziert, daß die Anzahl der Fehler, die erforderlich ist, um eine 1-Kette mit gegen ∞ divergierender Länge zu zerstören, ebenfalls gegen ∞ divergiert. Somit gilt $h(n) > r + 2$ für hinreichend großes n. ■

Satz 3.4 fußt auf dem Zusammenspiel zweier Effekte: Zwar trägt Aktion 0 inmitten einer 1-Kette zunächst einen vergleichsweise hohen Nutzen ein, sobald aber auch nur ein Spieler diese Aktion imitiert, fällt dieses Nutzenniveau für $v > 2r - 1$ und $r > 1$ so sehr ab, daß die Weiterverbreitung der Aktion 0 mittels der Imitationsregel unmöglich ist. Die Nachbarschaftsstruktur bewirkt also, daß unkooperatives Verhalten sich hauptsächlich selbst schädigt und sehr schnell nicht mehr imitiert wird. Die Verbreitung kooperativen Verhaltens wird hingegen durch die Nachbarschaftsstruktur sogar gefördert, denn zusammenhängende Gruppen kooperativer Spieler unterstützen sich vornehmlich untereinander, wodurch die Verbreitung von Kooperation nicht die Grundlage ihres eigenen Erfolgs zerstört.

3.3.3.3 Hybride Dynamik

Im Hinblick auf die weitere Analyse besteht die Haupterkenntnis des vorherigen Abschnitts darin, daß imitationales Verhalten eine gewisse Asymmetrie zugunsten kooperativen Verhaltens erzeugt. Diese Asymmetrie erscheint allerdings zu stark ausgeprägt, da Kooperation auch langfristig sogar dann auftritt, wenn sie individuell irrational ist. Gerade langfristig sollten die Spieler aber intellektuell in der Lage sein, dominante Strategien zu erkennen und zu verwenden. In Reaktion darauf wird nun eine hybride Dynamik betrachtet, in der die Spieler nicht nur imitieren, sondern auch beste Antworten geben und somit die eben vermißte Fähigkeit zurückerlangen.[40] Empirischen Rückhalt bekommt

[40] Eine Erweiterung des Modells von Eshel et al. (1998) mit einer ähnlichen Grundidee hat Matros (2004) vorgelegt. Allerdings betrachtet er ein anderes Stufenspiel und endogenisiert die Wahl der Entscheidungsregel in einer Weise, daß die Beste-Antwort-Regel keine echte Wirkung entfalten kann, da sie von kooperativen Spielern sogar langfristig niemals benutzt wird.

dieses Vorgehen von Huck, Normann und Oechssler (1999), die das Ergebnis eines von ihnen durchgeführten Experiments wie folgt beschreiben:

> If subjects have the necessary information to play best replies, most do so, though adjustment to the best reply is almost always incomplete. If subjects additionally have the necessary information to "imitate the best", at least a few subjects become pure imitators.

Sei also nun die Zahl der Entscheidungsregeln $m = 3$ und die verwendeten Regeln $d_1 = \mathcal{I}$, $d_2 = \mathcal{B}$ und $d_3 = \mathcal{T}$. Man beachte, daß alle nachfolgend beschriebenen Ergebnisse auch für den Fall gelten, daß die Imitationsregel nur mit sehr kleiner Wahrscheinlichkeit verwendet wird (kleine $\mu_1^1, \ldots, \mu_1^n > 0$). Insofern kann die hybride Dynamik als eine minimale Variation der Beste-Antwort-Dynamik gelten.

Die absorbierenden Zustände der unperturbierten hybriden Dynamik ergeben sich unmittelbar aus denen der zwei vorangegangenen. Sie werden im folgenden Satz benannt:

Satz 3.5 *Die unperturbierte hybride Dynamik hat immer den absorbierenden Zustand* $z^0 = (0, \ldots, 0)$. *Gilt* $s > 2r$, *dann ist zudem* $z^1 = (1, \ldots, 1)$ *absorbierend. Alle anderen Zustände sind transient.*

Beweis Offensichtlich ist ein Zustand unter der hybriden Dynamik genau dann absorbierend, wenn er unter der Beste-Antwort- *und* unter der Imitationsdynamik absorbierend ist. Dieser Satz folgt daher unmittelbar aus den Sätzen 3.1 und 3.3. ■

Während die hybride Dynamik kurzfristig der Beste-Antwort-Dynamik gleicht, unterscheiden sich die langfristigen Vorhersagen deutlich. Denn der Zustand z^1, in dem sich alle Spieler an der offenen Entwicklergemeinschaft beteiligen, ist unter bestimmten Voraussetzungen selbst dann stabil, wenn die durch den Parameter ζ gemessene strategische Unsicherheit sehr groß ist. Der folgende Satz präzisiert diese Aussage:

Satz 3.6 *Der Zustand z^1 ist der einzige stochastisch stabile Zustand der perturbierten hybriden Dynamik, wenn $s > 2r$, $r > 1$ und $v > (r+1)/(r-1)$ gilt und n hinreichend groß ist.*

Beweis Nach Satz 3.5 ist z^1 für $s \leq 2r$ nicht absorbierend und somit erst recht nicht stochastisch stabil. Die Bedingung $s > 2r$ muß also zwingend erfüllt sein. Gilt sie, so hat der unperturbierte Markow-Prozeß exakt die beiden Rekurrenzklassen $\{z^0\}$ und $\{z^1\}$. Wie bereits im Beweis von Satz 3.2 entspricht dann deren stochastisches Potential jeweils der Anzahl an Fehlern, die mindestens erforderlich ist, um die Rekurrenzklassen ausgehend von der anderen zu erreichen. Nachfolgend wird eine obere Schranke für die minimale Fehlerzahl bei einem Wechsel zu z^1 und eine untere Schranke für die minimale Fehlerzahl bei einem Wechsel zu z^0 bestimmt. Anschließend wird gezeigt, daß die letztgenannte Schranke für hinreichend großes n die erstgenannte übersteigt.

Zunächst wird der Wechsel von z^0 nach z^1 betrachtet. Hierbei nehme man an, im Zustand z^0 wechselten $l \in \mathbb{N}$ unmittelbar benachbarte Spieler aufgrund von Fehlern zur Aktion 1. Das Ergebnis ist ein Zustand a der folgenden Gestalt:

$$\ldots \quad 0 \quad \ldots \quad \overset{\alpha}{0} \quad \underbrace{1 \quad \ldots \quad 1 \quad \ldots \quad 1}_{l} \quad 0 \quad \ldots \quad 0 \quad \ldots$$

Die 1-Kette kann genau dann wachsen, wenn Spieler α Aktion 1 imitiert. Dieses ist sicher der Fall, wenn

$$\pi_1(a,\alpha) \geq \min\{l-1, 2r-1\}(1+v) > \pi_0(a,\alpha) = \min\{l,r\}(1+v) + 2r \quad (3.14)$$

gilt. Es ist nun leicht zu überprüfen, daß für $r > 1$ und $v > (r+1)/(r-1)$ der kleinste Wert für l, der Ungleichung (3.14) erfüllt, $l^* = \left\lfloor r + \frac{2r}{1+v} + 2 \right\rfloor$ ist. Wenn nun aber eine 1-Kette der Länge l^* wachsen kann, so kann es eine längere 1-Kette erst recht. Dieses Wachstum kann nun durch die Imitationsregel allein so lange voranschreiten, bis die 0-Kette auf die stabile Länge l' geschrumpft ist, die wegen $r > 1$ und $v > (r+1)/(r-1)$ die Bedingung $l' < r+1$ erfüllt. Zwar kann diese 0-Kette nicht durch die Imitationsregel eliminiert werden, um den Übergang nach

z^1 zu vollenden. Für hinreichend großes n ist dieses aber mithilfe der Beste-Antwort-Regel möglich, da dann $n - l' \geq n - r \geq \hat{n} = \lfloor \zeta n \rfloor$ gilt und Aktion 1 den Gemeinschaftsnutzen $s > 2r$ einbringt. Folglich erfüllt das stochastische Potential von $\{z^1\}$, nachfolgend mit ρ_{z^1} bezeichnet, für hinreichend großes n die Ungleichung

$$\rho_{z^1} \leq l^* = \left\lfloor r + \frac{2r}{1+v} + 2 \right\rfloor . \tag{3.15}$$

Nun wird der Wechsel von z^1 nach z^0 analysiert. Aus dem Beweis von Satz 3.2 ist bekannt, daß $n - \hat{n} + 1$ Fehler erforderlich sind, um allein mittels der Beste-Antwort-Dynamik zum Zustand z^0 zu wechseln. Die entscheidende Frage ist nun, ob imitationales Verhalten diese Anzahl verringern kann, und zwar mittels eines Ansteckungseffekts, der dem im vorherigen Abschnitt beschriebenen ähnelt. Nachfolgend wird daher die Rolle der Imitationsregel beim Wechsel zu z^0 für den Fall analysiert, daß die Entwicklergemeinschaft hinreichend groß ist und Aktion 1 stets den Gemeinschaftsnutzen s einträgt.

Ein Spieler i mit $a_i = 0$ heiße *isoliert*, wenn der kleinste Abstand zwischen ihm und einem anderen Spieler $j \neq i$ mit $a_j = 0$ größer als $2r+1$ ist.[41] Er liegt dann zwischen zwei 1-Ketten, deren Länge jeweils mindestens $2r + 1$ beträgt. Offensichtlich findet nun ein isolierter Spieler α, der fehlerhaft zur Aktion 0 wechselt, keinen Nachahmer, da alle Spieler in seiner Nachbarschaft $N(\alpha)$ in ihrer Nachbarschaft einen Spieler haben, der Aktion 1 verwendet und wegen $s > 2r$ den höchstmöglichen Nutzen $(1+v)2r+s$ erhält. Daher dürfen zwei Spieler, die fehlerhaft zur Aktion 0 wechseln, nicht isoliert sein, um möglicherweise Nachahmer finden zu können. Folglich führen ausgehend von z^1 zwei Fehler zu maximal $2r + 2$ Spielern mit Aktion 0 und allgemein k Fehler zu maximal $(k-1)2r + k$ Spielern, die Aktion 0 verwenden. Da diese $(k-1)2r+k$ Spieler eine 1-Kette bilden, deren Länge die einer stabilen 1-Kette übersteigt, werden sie keine weiteren Nachahmer finden. Die verbleibenden Spieler müssen daher mittels der Beste-Antwort-Regel zur Aktion 0 wechseln, um den Wechsel zu z^0 zu vollenden. Die-

[41] Aufgrund der kreisförmigen Anordnung der Spieler ist der Abstand zwischen zwei Spielern $i, j \in I$ mit $i \leq j$ als $\min\{j - i, n + i - j\}$ definiert.

ses erfordert, daß der Gemeinschaftsnutzen s nicht mehr ausgezahlt wird, was erst dann der Fall ist, wenn $(k-1)2r+k > n-\hat{n}$ gilt. Die kleinste Fehleranzahl $k \in \mathbb{N}$, welche diese Ungleichung erfüllt, ist $k^* = \lfloor (n-\hat{n}+2r)/(2r+1)+1 \rfloor$. Das stochastische Potential von $\{z^0\}$, nachfolgend mit ρ_{z^0} bezeichnet, erfüllt damit für hinreichend großes n die Ungleichung

$$\rho_{z^0} \geq k^* = \left\lfloor \frac{n - \lfloor \zeta n \rfloor + 2r}{2r+1} + 1 \right\rfloor . \tag{3.16}$$

Offensichtlich divergiert ρ_{z^0} nach Ungleichung (3.16) für $n \to \infty$ ebenfalls gegen ∞, wohingegen ρ_{z^1} nach Ungleichung (3.15) nach oben beschränkt ist. Folglich ist z^1 für hinreichend großes n der einzige stochastisch stabile Zustand. ■

Wie bereits in Abschnitt 3.3.3.2 entfaltet also Imitation unter bestimmten Voraussetzungen mehr Schubkraft beim Übergang von z^0 nach z^1 als vice versa. Dabei macht die Bedingung $s > 2r$ den Zustand z^1 gegenüber der Beste-Antwort-Regel robust. Die Bedingungen $r > 1$ und $v > (r+1)/(r-1)$ stellen zusammen sicher, daß eine hinreichend lange 1-Kette Nachahmer findet, selbst wenn dieses vorübergehend irrational ist. Und schließlich gibt die Forderung nach einem hinreichend großen n der durch die Imitationsregel eingeführten Asymmetrie zugunsten kooperativen Verhaltens größeres Gewicht.

Satz 3.6 beschreibt das zentrale Ergebnis des vorliegenden Modells. Es scheint daher angebracht, seine Robustheit gegenüber Veränderungen der Modellannahmen zu diskutieren. So mag man einwenden, die Spezifikation des Gemeinschaftnutzens $\sigma(a)$ sei zu simpel oder sogar kontrafaktisch, da einige Motive für die Beteiligung an offenen Softwareprojekten (z. B. Job-Market-Signaling) möglicherweise an Kraft verlieren, sobald die Gemeinschaft zu groß wird. Es wäre dann angemessener, diese Nutzenkomponente durch eine allgemeine Funktion $\sigma' : [0,1] \to \mathbb{R}$ der relativen Gemeinschaftsgröße $g(a)/n$ zu beschreiben, die erst monoton steigt und dann monoton fällt. Diese allgemeinere Beschreibung kann aber leicht auf die hier betrachtete abgebildet werden, indem $s = \sigma'(1)$ und $\zeta = \min\{z \in (0,1) \,:\, \sigma'(y) > s \text{ für alle } y \geq z\}$ gesetzt wird. Die die langfristige Stabilität von z^1 sichernde Anforderung ist allein, daß der Gemeinschaftsnutzen

eine untere Schranke $s > 2r$ übersteigt, sobald die Gemeinschaft eine gewisse relative Größe $\zeta < 1$ erreicht hat.

Überdies ist auch eine alternative Modellierung der Trägheit denkbar. Einerseits könnte etwa in jeder Periode genau ein Spieler seine Aktion überprüfen. Dieses würde insbesondere den gleichzeitigen Wechsel vieler Spieler mittels einer Entscheidungsregel unterbinden; stattdessen müßten die Spieler hintereinander und in einer jeweils veränderten Situation wechseln. Andererseits könnte aber auch Trägheit überhaupt keine Rolle spielen. Es ist aber leicht zu zeigen, daß Satz 3.6 für beide Alternativen ebenfalls gilt.

Bergin und Lipman (1996) kritisieren das Konzept stochastischer Stabilität ganz grundsätzlich. Die alleinige Berücksichtigung der Anzahl der erforderlichen Fehler für Übergänge zwischen Rekurrenzklassen greife zu kurz. Denn sobald die Wahrscheinlichkeiten, mit der diese Fehler auftreten, für $\varepsilon \to 0$ verschieden schnell gegen Null konvergieren, verändern sich die langfristigen Eigenschaften der perturbierten Markow-Prozesse erheblich. Allerdings gibt es keinen Grund, warum die Fehlerwahrscheinlichkeiten im hier vorliegenden Modell verschiedene Konvergenzraten aufweisen sollten. Zudem zeigen van Damme und Weibull (2002), wie die Konvergenzraten sinnvoll endogenisiert werden können, um das Konzept stochastischer Stabilität zu rehabilitieren.

Eine andere offensichtliche Schwäche der hier verwendeten Methoden liegt darin, daß alle Ergebnisse nur auf unendlich lange Sicht gelten. Es ist daher schwierig einzuschätzen, inwiefern diese Ergebnisse bereits innerhalb nach menschlichem Maßstab relevanter Zeitspannen Geltung haben. Es ist aber allgemein bekannt, daß lokale Interaktionsstrukturen die Konvergenzgeschwindigkeit solcher Modelle wie dem hier beschriebenen drastisch erhöhen (vgl. etwa Ellison, 1993). Insofern sollte dieser allgemeinen Schwäche hier allenfalls geringe Bedeutung zukommen.

3.3.4 Schlußfolgerungen

Das Hauptanliegen des hier entwickelten Modells besteht in der Klärung der Frage, unter welchen Voraussetzungen sich offene Entwicklergemeinschaften nachhaltig und zuverlässig etablieren können. Satz 3.6 benennt diesbezüglich vier hinreichende Bedingungen:

- Der Nutzen aus der Beteiligung an einer großen, funktionierenden offenen Entwicklergemeinschaft muß den aus dem Verkauf der einzelner Module übersteigen ($s > 2r$). Anderenfalls bricht die Kooperation aufgrund von Eigeninteressen zusammen.

- Die Kopplung zwischen den Modulen muß hinreichend stark sein($r > 1$). Die einzelnen Bausteine der Software dürfen also nicht lose nebeneinander stehen, sondern müssen ein zusammenhängendes Softwaresystem bilden.

- Die Verfügbarkeit der Quelltexte muß hinreichend wertvoll sein($v > (r + 1)/(r-1)$). Offensichtlich ist diese Bedingung tendenziell eher bei Software für technisch versierte Nutzer erfüllt.

- Die Software muß hinreichend stark modularisiert sein (großes n). Einzelnen darf also keine zu große Bedeutung zukommen, damit diese ein kooperatives Regime nicht quasi im Alleingang absetzen können.

Zudem kann das Modell nur dann als adäquate Beschreibung der Realität gelten, wenn die zwei folgenden Voraussetzungen erfüllt sind:

- Die Software besteht aus wohldefinierten Modulen mit einer klaren Aufgabenverteilung. Dieses ist insbesondere in reifen Softwarekategorien gegeben.

- Es existiert ein effizienter Kommunikationsmechanismus, der den Entwicklern wiederholte und friktionsfreie Interaktion ermöglicht. Im Fall offener Softwareentwicklung ist dieser Mechanismus natürlich das Internet.

All diese Bedingungen werden in der Realität von erfolgreichen offenen Softwareprojekten regelmäßig erfüllt. Sollte man also erwarten, daß offene Entwicklergemeinschaften in Zukunft wahrlich innovative, aber auch kleine und monolithische Software für technisch inkompetente Nutzer hervorbringen? Wahrscheinlich nicht! Sollte man hingegen erwarten, daß das Betriebssystem GNU/Linux, Ergebnis der Bestrebungen Tausender von IT-Profis, das wohlbekannte, hochmodulare Unix zu reproduzieren, seine Erfolgsgeschichte fortschreibt. Im Licht des hier vorgestellten Modells ist die Antwort "Ja"!

3.4 Zwischenfazit

Entgegen der populären Ansicht, offene Software sei im wesentlichen das Ergebnis altruistischen Strebens, kann man ihre Entwicklung auf handfeste ökonomische Motive zurückführen. Diese sind allerdings derartig vielfältig, daß eine auf ein Motiv beschränkte Betrachtung die Funktionsweise einer offenen Entwicklergemeinschaft nur begrenzt zu erfassen vermag. Ein Ansatzpunkt für eine umfassendere Analyse erwächst aber aus der Beobachtung, daß alle Motive nur wirksam sind, wenn die Entwicklergemeinschaft eine gewisse Mindestgröße erreicht hat. Offene Softwareentwicklung stellt sich dann in erster Linie als Koordinationsproblem zwischen vielen Akteuren dar. Eine genaue Analyse dieses Koordinationsproblems ermöglicht Aussagen darüber, unter welchen Voraussetzungen sich offene Software langfristig etablieren kann.

4 Offene Software und staatliches Handeln

Auch in der politischen Arena hat offene Software in vielfältiger Weise für Aufsehen gesorgt. Weltweit sind auf multinationaler, nationaler und regionaler Ebene zahlreiche staatliche Initiativen und Gesetzesvorhaben vorgelegt oder bereits umgesetzt worden, die offene Software in verschiedenster Weise fördern sollen. Die Motive sind dabei vielfältig. So wird beispielsweise argumentiert, offene Software sei proprietärer technisch überlegen oder geeignet, monopolistische Marktstrukturen aufzubrechen und den Wettbewerb zu fördern. Andere sehen offene Software als regelrechten Innovationsmotor. Ob offene Software aus diesen und anderen Motiven heraus überhaupt staatlich gefördert werden sollte und wie dieses gegebenenfalls zu geschehen habe, ist allerdings umstritten. Handfeste wirtschaftliche Interessen auf der einen und ein fast schon missionarischer Eifer auf der anderen Seite haben dieser Debatte zudem einen bisweilen recht schrillen Ton gegeben. So äußerte sich Jim Allchin, Vizepräsident von Microsoft, öffentlich wie folgt (zitiert nach O'Reilly, 2001):

> I can't imagine something that could be worse than this for the software business and the intellectual-property business. [...] I'm an American, I believe in the American Way. I worry if the government encourages open source, and I don't think we've done enough education of policy makers to understand the threat.

Auf der anderen Seite empfiehlt das President's Information Technology Advisory Committee (2000) unter der Leitung von Irving Wladawsky-Berger, dem

Vizepräsidenten für Technologie und Strategie von IBM und langjährigem Verfechter offener Software:

> First, the Federal government should aggressively encourage the development of open source software for high end computing. [...] Second, a "level playing field" must be created within the government procurement process to facilitate open source development.

Auch in der deutschen Parteienlandschaft hat sich diese Debatte niedergeschlagen. Welch vielfältige Positionen dabei vertreten werden, belegen die folgenden Zitate. So heißt es in einem Antrag von Mitgliedern der Bundestagsfraktionen von SPD und Bündnis 90/Die Grünen (2001):

> Der Deutsche Bundestag begrüßt die Förderung von Open-Source-Produkten und fordert die Einführung von unter Open-Source-Lizenz erstellten Produkten in der Bundesverwaltung, vor allem in sicherheitsrelevanten Bereichen.

In einem Beschluß des Bundesvorstands der CDU (2002) wird die folgende, vermutlich überraschende Haltung formuliert:

> Zur Förderung von Open-Source-Software sollten öffentliche Stellen nur Software verwenden, deren Quellcode frei zugänglich ist, soweit solche Software verfügbar ist.

Die Bundestagsfraktion der FDP (2002) steht einer staatlichen Förderung offener Software hingegen eher ablehnend gegenüber:

> Auch die FDP-Fraktion begrüßt grundsätzlich den Einsatz von Open-Source Produkten in der Verwaltung. [...] Was wir kritisieren, sind Linux-Anwendungen, die sich über die objektiv nachprüfbaren Kriterien Bedienerfreundlichkeit, Stabilität und Kosten hinwegsetzen, um politisch erwünschte Signalwirkungen zu erreichen.

Und schließlich äußert sich die Bundestagsfraktion der PDS (2001) mit vertrautem Klang wie folgt:

> Die PDS setzt sich für Open Source Software und gegen Softwarepatente ein. Die Offenlegung der Quellcodes ermöglicht eine Vergesellschaftung der Programme und damit ihre stetige Verbesserung und Verbreitung.

In diesem Kapitel soll versucht werden, die oben skizzierte Debatte zu strukturieren und zu versachlichen. Dazu werden zunächst die Maßnahmen und Instrumente, mit denen die öffentliche Hand offene Software fördern kann, beschrieben. Anschließend werden die ökonomischen Auswirkungen staatlichen Handelns auf Märkte für Software analysiert, wobei der Schwerpunkt auf solche für Massensoftware gelegt wird. Es folgt eine Untersuchung der Bedeutung offener Software für die Entwicklung von Individualsoftware. Schließlich werden sämtliche Zwischenergebnisse kritisch gewürdigt und einige Empfehlungen für staatliches Handeln in bezug auf offene Software gegeben.

4.1 Maßnahmen und Instrumente

Die bislang weltweit diskutierten oder implementierten Maßnahmen und Instrumente zur Förderung offener Software sind tatsächlich äußerst vielfältig, lassen sich aber dennoch im wesentlichen in vier Gruppen einteilen: nachfrageseitige Förderung, angebotsseitige Förderung, Information und Beratung sowie die Schaffung eines neutralen Wettbewerbsumfelds.

4.1.1 Nachfrageseitige Förderung

Selbstverständlich tritt die öffentliche Hand auch als Nachfrager nach Software auf und kann offene Software somit über deren *Nutzung in öffentlichen Einrichtungen* fördern, indem sie diese verwendet.[1] Von einem Instrument zur

[1] Nach Angaben des Statistischen Bundesamts waren im Jahr 2002 knapp 15 % aller abhängig Beschäftigten im öffentlichen Dienst tätig. Unterstellt man, daß sich die Nachfrage nach

Förderung kann allerdings erst dann gesprochen werden, wenn die Entscheidung für offene Software nicht allein auf der Grundlage einer neutralen Abwägung von Kosten und Nutzen erfolgt, sondern aufgrund eines artikulierten politischen Willens auch in solchen Fällen, in denen allein unter Kosten-Nutzen-Aspekten einer proprietären Software der Vorzug hätte gegeben werden müssen. Schwach ausgestaltet fordert dieses Instrument von öffentlichen Einrichtungen, offener Software den Vorzug vor proprietärer Software zu geben, wenn diese im Prinzip gleichwertig sind. Stark ausgestaltet erzwingt es hingegen, offene Software einzusetzen, wenn sie verfügbar ist. In der letztgenannten Ausgestaltung wurde dieses Instrument zwar von den gesetzgebenden Organen vieler Länder diskutiert, allerdings nie beschlossen (vgl. Center for Strategic and International Studies, 2004); in der erstgenannten Ausgestaltung wird es jedoch in mehreren Ländern genutzt, beispielsweise in Brasilien und Malaysia (vgl. Heise-Newsticker, 2003a, 2004).

Des weiteren kann ein Staat die Nachfrage nach offener Software stimulieren, indem er ihre *Nutzung finanziell subventioniert*. Außer Singapur verfolgt allerdings kein Land eine derartige Politik (vgl. Hahn, 2002) .

In Deutschland wurde die Förderung offener Software durch eine Stärkung der Nachfrage zwar diskutiert, etwa im Deutschen Bundestag (2001), aber schließlich verworfen.

4.1.2 Angebotsseitige Förderung

Angebotsseitig kann die öffentliche Hand offene Software fördern, indem sie deren Entwicklung unterstützt. Dieses kann auf drei Arten geschehen.

Erstens kann die Weiterentwicklung bereits vorhandener offener Software gefördert werden, etwa durch die Zahlung von *Subventionen* an deren Entwickler. Ein praktisches Problem bei der Anwendung dieses Instruments kann in einer

Software pro Arbeitnehmer im öffentlichen nicht wesentlich von der im privaten Sektor unterscheidet, erhält man eine Vorstellung davon, über welchen Anteil der gesamten Nachfrage nach Software die öffentliche Hand gebietet.

starken Fragmentierung der Entwicklergemeinschaft bestehen, die es eventuell unmöglich macht, einen geeigneten Adressaten für die Subventionen zu identifizieren. Alternativ können sich allerdings auch öffentliche Einrichtungen, die über Beschäftigte mit entsprechenden Fachkenntnissen verfügen, unmittelbar an der Entwicklung beteiligen. Neben zahlreichen anderen Ländern engagiert sich auch Deutschland in dieser Weise für offene Software; beispielsweise beteiligt sich das Bundesministerium des Innern (2002) an der Entwicklung offener Sicherheitssoftware.

Das eben beschriebene Vorgehen steigernd kann die öffentliche Hand zweitens die Entwicklung offener Software *initiieren* und dann maßgeblich vorantreiben. Den Anstoß dafür gibt häufig der Eigenbedarf einer öffentlichen Einrichtung, der in Ermangelung geeigneter Massensoftware nur mit Individualsoftware befriedigt werden kann. Die so entstehende Software wird dann unter eine offene Lizenz gestellt, um sie der Allgemeinheit so nutzbringend wie möglich zur Verfügung zu stellen. Gelegentlich sind bei dieser recht aggressiven Form angebotsseitiger Förderung sicherlich auch industriepolitische Motive von Bedeutung. Ein Beispiel für diese Politik ist die Kooperation Chinas, Japans und Südkoreas bei der Entwicklung einer "asiatischen" Linux-Variante (vgl. Heise-Newsticker, 2003b).

Zudem entwickelt die öffentliche Hand Software, da sie etwa an Universitäten entsprechende Grundlagenforschung betreibt. Indem sie vorschreibt, die Ergebnisse dieser Forschung als offene Software[2] zu veröffentlichen, verfügt sie über eine dritte Möglichkeit der angebotsseitigen Förderung.

4.1.3 Information und Beratung

Des weiteren kann der Staat offene Software mit Hilfe von Informations- und Beratungsangeboten fördern. Die zu beobachtende Bandbreite derartiger Ange-

[2] In der Literatur besteht weitgehend Einigkeit darüber, daß die Ergebnisse öffentlich finanzierter Forschung nicht unter eine Copyleft-Lizenz gestellt werden sollten, um eine möglichst weitreichende und somit auch kommerzielle Weiterverwendung der Software nicht zu behindern (vgl. etwa Evans, 2002; Smith, 2002; Schmidt und Schnitzer, 2003).

bote ist dabei erheblich und reicht von Initiativen, die beinahe den Charakter von Werbekampagnen aufweisen, bis zur Bereitstellung umfangreicher Leitfäden mit ausführlichen Anleitungen zum konkreten Einsatz offener Software.

Wie eine umfangreiche Untersuchung des Center for Strategic and International Studies (2004) zeigt, werden derartige Maßnahmen in zahlreichen Ländern weltweit und auch in Deutschland intensiv und sehr häufig eingesetzt.

4.1.4 Schaffung eines neutralen Wettbewerbsumfelds

Schließlich kann die öffentliche Hand Rahmenbedingungen schaffen, die den Besonderheiten offener Software Rechnung tragen und so einen unverzerrten Wettbewerb mit proprietärer Software ermöglichen. Drei Bereiche sind dabei hervorzuheben:

Eine hitzige Debatte ist um die Frage entbrannt, ob die Patentierbarkeit von Algorithmen und Datenstrukturen, die ja den Kern jeder Art von Software bilden, eine systematische, gesamtgesellschaftlich unerwünschte Benachteiligung offener Software gegenüber proprietärer zur Folge hat. Im wesentlichen werden zwei mögliche Asymmetrien genannt, die beide daraus erwachsen, daß die Durchsetzung eigener Rechte gegen eine finanziell deutlich stärkere Partei schwierig ist (vgl. Richter, 2003). Erstens wird befürchtet, die Entwickler offener Software könnten sich nicht gegen die Patentierung ihrer eigenen, offen gelegten Entwicklungen durch finanzkräftige Unternehmen mit entsprechenden Rechtsabteilungen wehren; zweitens sei es Unternehmen durch zwar nicht gerichtsfeste aber zahlreiche Patente auf Trivialitäten[3] möglich, die Entwicklung konkurrierender offener Software faktisch zu unterbinden. Des weiteren wird häufig argumentiert, offene Lizenzen und Softwarepatente seien juristisch inkompatibel.

Man kann nun argumentieren, Patente seien ein derartig wichtiges Instrument zur Versöhnung privater und gesamtgesellschaftlicher Innovationsanreize, daß mögliche negative Effekte auf die Entwicklung offener Software vernachlässigt werden können. Andererseits wird gerade diese Wirkung von Patenten speziell

[3] Man spricht in diesem Zusammenhang auch von Patentdickichten.

im Fall von Software zunehmend bestritten: Da die Softwareindustrie durch inkrementelle und komplementäre Innovationen sowie eine sehr hohe Rate der Wiederverwendung von Quelltexten gekennzeichnet sei, würden Patente sich insgesamt eher innovationshemmend auswirken.[4] Insgesamt geht die Frage, ob Software patentierbar sein sollte, also über das Thema "Offene Software" weit hinaus. Sie soll daher im Rahmen dieser Arbeit nicht weiter diskutiert werden.

Eine weitere Verzerrung wider offene Software kann aus *Zertifizierungserfordernissen* bei der Vergabe öffentlicher Aufträge entstehen. Gerade für solche Anwendungsfelder, in denen Stabilität und Sicherheit der eingesetzten Software höchsten Maßstäben genügen müssen, ziehen öffentliche Einrichtungen regelmäßig nur Angebote in Betracht, denen eben jene Qualitätsmerkmale nach umfangreichen Tests von unabhängigen oder auch staatlichen Zertifizierungsstellen bescheinigt werden. Sind nun die teilweise erheblichen Kosten der Zertifizierung vom Anbieter zu tragen, kann dieses allein den Ausschluß offener Software von der Vergabe bedeuten.[5] Es mag daher sinnvoll sein, die Richtlinien für die Vergabe von öffentlichen Aufträgen entsprechend anzupassen.

Zudem werfen Lizenzen für offene Software aufgrund ihrer unkonventionellen Nutzung des Urheberrechts zahlreiche juristische Fragen etwa in den Bereichen des Patent-, Marken-, Wettbewerbs- und des Vertragsrechts auf.[6] Die daraus für Entwickler und Anwender gleichermaßen erwachsende *Rechtsunsicherheit* stellt zweifelsohne ein Hemmnis für den Erfolg offener Software dar, kann aber vom Gesetzgeber und der Rechtsprechung erheblich vermindert werden.

---

[4] Eine theoretische sowie eine empirische Untersuchung mit diesem Ergebnis sowie Überblicke über die entsprechende Literatur finden sich bei Bessen und Maskin (2000) bzw. Bessen und Hunt (2004).

[5] Wird die jeweilige offene Software hingegen von hinreichend finanzkräftigen Unternehmen getragen, verliert dieses Problem an Bedeutung. So haben IBM und SuSE die von ihnen angebotene Variante von Linux EAL2-zertifizieren lassen, um den Weg für die Migration der Stadt München auf eben dieses Betriebssystem freizumachen; die Kosten für eine derartige Zertifizierung betragen ca. 400.000 € (vgl. Shankland, 2003).

[6] Für das deutsche Recht beschreiben und analysieren Jaeger und Metzger (2002) diese Fragen sehr ausführlich.

4.2 Märkte für Massensoftware

In diesem Abschnitt wird die Wirkung der oben beschriebenen Maßnahmen und Instrumente auf Märkte für Massensoftware[7] behandelt. Unter Massensoftware soll dabei solche Software verstanden werden, die ohne individuelle Anpassungen von zahlreichen Konsumenten verwendet wird; Beispiele wie etwa Microsoft Windows und Office dürften jedem aus der eigenen Nutzung bestens vertraut sein.

Die Analyse folgt dabei dem üblichen industrieökonomischen Vorgehen. Zunächst werden der betrachtete Markt, in diesem Fall also der Markt für Massensoftware, sowie die Möglichkeiten staatlicher Einflußnahme in ein geeignetes Modell übertragen und das Partialmarktgleichgewicht[8] bestimmt. Dann wird in einer komparativ-statischen Analyse die Wirkung staatlicher Eingriffe auf die gleichgewichtige Produzenten- und Konsumentenrente sowie die gesamtgesellschaftliche Wohlfahrt bestimmt.

4.2.1 Eigenschaften von Märkten für Massensoftware

Die folgende Analyse muß den besonderen Eigenschaften von Märkten für Massensoftware Rechnung tragen. Diese ergeben sich aus der einzigartigen Doppelrolle von Software, sowohl ein Informationsgut als auch eine Informationstechnologie zu sein.

Nach Varian (1998) sind bei Märkten für Informationsgüter drei Aspekte hervorzuheben: Skaleneffekte in der Produktion, Freiräume bezüglich der Gestaltung von Rivalität und Ausschließbarkeit im Konsum sowie der Erfahrungsgutcharakter der gehandelten Güter.

Erhebliche *Skaleneffekte* entstehen, da die Produktion von Informationen üblicherweise teuer, ihre Reproduktion hingegen billig ist. Die digitale Revolution

[7] Branchenübliche Bezeichnungen für derartige Software sind auch in Deutschland "packaged software", "shrink-wrapped software" und "commercial-off-the-shelf (COTS) software".

[8] Existieren multiple Gleichgewichte, so müssen sie selbstverständlich alle betrachtet werden.

um das Internet hat diesen Effekt noch verstärkt. So kann die Entwicklung einer komplexen Software leicht Fixkosten in Millionen- oder gar Milliardenhöhe verursachen, während ihre Verbreitung über das Internet nur wenige Cent kostet.

Der Aspekt, daß insbesondere Software in den beiden Dimensionen der *Rivalität und Ausschließbarkeit* über entsprechende Lizenzvereinbarungen sehr frei positioniert werden kann, wurde bereits am Anfang dieser Arbeit eingehend diskutiert. Offene Software ist ein öffentliches, proprietäre Software ein privates Gut.

Ein sogenanntes *Erfahrungsgut* ist nach Nelson (1970) dadurch gekennzeichnet, daß ein Konsument dessen Wert erst kennt, nachdem er es benutzt hat. Handelt es sich bei diesem Gut um Informationen, spitzt sich dieser Aspekt zum "Informationsparadoxon" zu, dem zufolge ein Konsument seine Zahlungsbereitschaft für Informationen erst nach deren Offenbarung festlegen kann, dann aber nur schwer zur Zahlung zu bewegen ist (vgl. Arrow, 1962). Für offene Software löst sich dieses Paradoxon auf; von Herstellern proprietärer Software kann es etwa über den Aufbau einer reputierlichen Marke oder die kostenlose Abgabe im Funktionsumfang reduzierter Versionen zu Demonstrationszwecken im Regelfall gut kontrolliert werden. Daher werden Effekte, die aus dem Erfahrungsgutcharakter von Software erwachsen, nachfolgend vernachlässigt.

Des weiteren weisen Märkte für Massensoftware regelmäßig all jene Merkmale auf, die solchen für Informationstechnologien ganz allgemein zu eigen sind; Shapiro and Varian (1999) heben in diesem Zusammenhang Netzwerkeffekte, Wechselkosten und Systemwettbewerb hervor.

Sogenannte *Netzwerkeffekte* liegen vor, wenn der Nutzen, den ein Gut einem Konsumenten stiftet, umso größer ist, je mehr weitere Konsumenten sich für eben jenes Gut entschieden haben. Ein Paradebeispiel ist der Markt für sogenannte Instant Messenger Software, die es ihren Nutzern ermöglicht, in Echtzeit zu kommunizieren und Daten auszutauschen. Es ist offensichtlich, daß der Nutzen einer derartigen Software quasi ausschließlich von der Anzahl der potenti-

ellen Kommunikationspartner abhängt.[9] Die Literatur zu Netzwerkeffekten ist inzwischen außerordentlich vielfältig. Auf der einen Seite haben Katz und Shapiro (1985) sowie Farrell und Saloner (1985) aufbauend auf eine frühe Arbeit von Rohlfs (1974) eine Lawine theoretischer Arbeiten ausgelöst, die verschiedene aus Netzwerkeffekten erwachsende Phänomene untersuchen, beispielsweise das Problem der kritischen Masse an Konsumenten bei der Markteinführung eines neuen Netzwerkgutes, die Multiplizität von Marktgleichgewichten bei starken Netzwerkeffekten sowie die daraus resultierenden Fragen nach Koordination und Effizienz, die Kompatibilität zwischen konkurrierenden Netzwerken sowie die Bedeutung einer installierten Basis und der daraus resultierenden Pfadabhängigkeiten bei der Diffusion von Netzwerkgütern. Eine gute Übersicht über diese Literatur vermitteln Katz und Shapiro (1994), Economides (1996), Shy (2001) und Gandal (2002). Auf der anderen Seite versuchen empirische Arbeiten, die Existenz von Netzwerkeffekten nachzuweisen und ihre Höhe insbesondere für die Bestimmung hedonischer Preisindizes zu schätzen. Gandal (1994), Brynjolfsson und Kemerer (1996), Gröhn (1999) sowie Yasaki und Murakami (2003) untersuchen dabei diverse Märkte für Massensoftware und finden klare Hinweise für die Existenz von Netzwerkeffekten.

Sogenannte *Wechselkosten* entstehen einem Konsumenten, wenn er den Anbieter eines Gutes wechselt. Ein Beispiel gibt jeder Markt für Bürosoftware. So muß ein Konsument etwa beim Wechsel einer Textverarbeitung seine bestehenden Dateien konvertieren und eine neue Benutzerschnittstelle erlernen; beides verursacht offensichtlich Kosten, die schnell erheblich werden können, wenn in einer größeren Organisation mehrere tausend Arbeitsplätze umzustellen sind. Wechselkosten in ausreichender Höhe können dazu führen, daß einem Konsumenten der Wechsel des Anbieters faktisch unmöglich ist; man bezeichnet die-

[9] Netzwerkeffekte waren auch die Ursache für den legendären "Instant Messenger War" zwischen America Online, Microsoft und Yahoo, in dem die Kontrahenten in einer Serie technischer Manöver wechselseitig versuchten, ihre Netzwerke gegen den Zugriff von außen abzuschotten und gleichzeitig ihre Kunden an die Netzwerke der Wettbewerber anzubinden (vgl. Wong und Junnarkar, 1999).

ses Phänomen auch als *Lock-In*. Die Möglichkeit des Lock-In von Konsumenten führt unmittelbar zu dem unternehmerischen Abwägungsproblem, welches den Kern jeder Analyse von Wechselkosten bildet. Es besteht darin, daß ein Unternehmen einerseits einen hohen Preis setzten möchte, um vom Lock-In bereits gewonnener Kunden zu profitieren, andererseits aber durch niedrige Preise eine große Zahl an Neukunden zu gewinnen wünscht. Wie dieses Abwägungsproblem gelöst wird und welche Auswirkungen es auf das Preisniveau, die Wettbewerbsintensität und die Struktur eines Marktes hat, untersucht eine gut entwickelte Literatur; eine Übersicht gibt Klemperer (1995). Die einzige empirische Arbeit, welche zumindest Indizien für die Existenz signifikanter Wechselkosten in einem Markt für Massensoftware findet, stammt von Yasaki und Murakami (2003). Sie finden einige Regelmäßigkeiten, die sich durch Wechselkosten erklären lassen. Zudem hat das Marktforschungsunternehmen Yankee Group (2004) in einer Umfrage unter 1.000 Unternehmen ermittelt, daß 90 % der Unternehmen mit mehr als 10.000 Arbeitsplätzen aufgrund prohibitiv hoher Wechselkosten nicht von Windows auf Linux umsteigen werden.

Systemwettbewerb liegt vor, wenn ein Konsument auf mehreren Märkten für komplementäre Güter, im folgenden Systemkomponenten genannt, agiert und Einschränkungen bezüglich der Kombinierbarkeit verschiedener Komponenten zu einem Gesamtsystem bestehen. Ein Beispiel bilden gemeinschaftlich die Märkte für Betriebssysteme und für Anwendungssoftware. Jeder Konsument ist letztlich an einem Gesamtsystem bestehend aus einem Betriebssystem und mehreren Anwendungen interessiert, kann aber, wenn er sich etwa für das Betriebssystem Linux entscheidet, keine Anwendungen benutzen, welche für Windows entwickelt wurden. Die so entstehenden Wechselbeziehungen zwischen Märkten werden in der Literatur ausführlich, aber auch uneinheitlich analysiert. Hervorzuheben sind der auf Matutes und Regibeau (1988) und Economides (1989) zurückgehende Mix-and-Match-Ansatz, welcher die Wechselbeziehung zwischen Systemkomponenten direkt modelliert und insbesondere Kompatibilitätsfragen behandelt, und die maßgeblich von Caillaud und Jullien (2001, 2003) und Rochet und Tirole (2002, 2003) initiierte Literatur zu zweiseitigen Märkten, die

sich unter anderem damit befaßt, welche Strategie der Anbieter einer Systemkomponente von zentraler Bedeutung sowohl gegenüber dem Konsumenten als auch den Anbietern anderer, komplementärer Systemkomponenten verfolgen sollte.

Es ist äußerst schwierig, alle in diesem Abschnitt beschriebenen Eigenschaften von Märkten für Massensoftware in einem einzigen Modell zu erfassen. Daher wird in dieser Arbeit der Weg gewählt, ein einfaches Modell als vereinheitlichende Grundlage zu nehmen und es anschließend in die oben vorgegebenen Richtungen zu erweitern. Eine Zusammenfassung aller Ergebnisse findet sich in Gutsche (2005a).

4.2.2 Grundmodell

Nachfolgend wird der Wettbewerb zwischen einer proprietären und einer offenen Software in einem Massenmarkt mit Hilfe eines Modells horizontaler Produktdifferenzierung à la Hotelling (1929) beschrieben. Zugrundegelegt wird wie üblich ein Kontinuum von Konsumenten, deren Typ θ auf dem Intervall $[0, 1]$ gleichverteilt sei. Die proprietäre und die offene Software haben die Positionen 0 bzw. 1. Diese Positionen werden als fix angenommen, sind also anders als in der Literatur zur Produktpositionierung nicht Gegenstand strategischer Erwägungen. Diese Annahme ist sinnvoll, da der Wettbewerb zwischen zwei Computerprogrammen untersucht werden soll, die sich allein in der Art der Lizenzierung unterscheiden, ansonsten aber vollkommen gleichartig sind.[10] Wären die konkurrierenden Programme hingegen in weiteren Dimensionen horizontal oder auch vertikal differenziert, würden möglicherweise weitere Effekte, die ausdrücklich nicht betrachtet werden sollen, die Ergebnisse überlagern. Die Positionierung erfaßt also den beiden Lizenzierungsarten jeweils immanente Eigenschaften und ist in dem Sinne tautologisch, daß die proprietäre Software die Position 0 hat, weil sie proprietär ist; gleiches gilt für die offene Software

[10] Zudem hat diese Annahme den erfreulichen Nebeneffekt, daß sie unstetige Gewinnfunktionen, wie sie d'Aspremont, Gabszewicz und Thisse (1979) beschreiben, vermeidet.

und Position 1. Ein Konsument nahe 1 schätzt dann etwa die Verfügbarkeit des Quellcodes oder die Herstellerunabhängigkeit besonders hoch, während ein Konsument nahe 0 beispielsweise besonders großen Wert auf die Haftung eines Herstellers für grobe Fahrlässigkeiten legt. Die Annahme der Gleichverteilung der Konsumententypen zwischen beiden Polen dient allein der Vereinfachung der Analyse; de facto wird gegenwärtig in nahezu allen Märkten für Massensoftware die Verteilung zugunsten proprietärer Software asymmetrisch sein.

Ebenfalls zur Vermeidung von Verzerrungen wird des weiteren für beide Wettbewerber die gleiche Kostenstruktur angenommen, nämlich identische konstante Grenzkosten, die ohne Beschränkung der Allgemeinheit auf 0 normiert werden, und fixe Entwicklungskosten, die unwiederbringlich verloren sind und damit vernachlässigt werden können. Die offene Software kann als öffentliches Gut lizenzgebührenfrei verwendet werden, wohingegen für die proprietäre Software einheitliche Lizenzgebühren in Höhe von l zu entrichten sind. Der Hersteller der proprietären Software kann also aufgrund asymmetrischer Informationen nicht preisdiskriminieren. Alle weiteren bei den Konsumenten für den Einsatz eines der beiden Programme anfallenden Kosten seien wiederum identisch und bereits im Grundnutzen b enthalten; wie üblich sei der Grundnutzen so groß, daß alle Konsumenten eines der beiden Programme kaufen.

Erwirbt nun ein Konsument vom Typ θ die proprietäre Software für Lizenzgebühren in Höhe von l, so sei sein Nutzen gegeben durch

$$u_P(\theta, l) = b - t \cdot \theta - l \tag{4.1}$$

mit t als Präferenzintensität bezüglich der durch die verschiedenen Lizenzierungsmodelle bedingten horizontalen Differenzierung; kauft er hingegen die offene Software, so sei sein Nutzen gegeben durch

$$u_O(\theta) = b - t \cdot (1 - \theta). \tag{4.2}$$

Die staatliche Förderung offener Software kann in diesem Modell formal als ein Zusatznutzen $s \geq 0$ abgebildet werden, den ein Konsument bei der Nutzung offener Software erhält. Ursächlich für diesen Zusatznutzen können dabei

verschiedene Maßnahmen von Informations- und Beratungsangeboten über die Weiterentwicklung der offenen Software bis hin zu direkten Subventionen sein. Somit wird für den Rest dieses Abschnitts die Nutzenfunktion (4.2) wie folgt modifiziert:

$$u_O(\theta) = b - t \cdot (1 - \theta) + s. \tag{4.3}$$

Dabei wird $s < t$ angenommen, damit der staatlich geschaffene Zusatznutzen nicht zu einer vollständigen Verdrängung der proprietären Software vom Markt führt.[11]

Des weiteren wird die zwangsweise Adoption offener Software durch öffentliche Einrichtungen in der Weise modelliert, daß einem Anteil $z \in [0, 1]$ aller Konsumenten die Nutzung offener Software vorgeschrieben wird und nur noch der verbleibende Anteil $1 - z$ seine Kaufentscheidung über eine Abwägung von Kosten und Nutzen treffen kann. In beiden Gruppen seien die Typen der Konsumenten wiederum gleichverteilt. Es wird also angenommen, daß der Staat den Zwang aufgrund asymmetrischer Informationen nicht typabhängig ausübt; er hat also keine besseren Informationen als der Hersteller der proprietären Software.

Der Typ des Konsumenten, der zwischen den beiden Angeboten indifferent ist, ergibt sich aus der Gleichsetzung von (4.1) und (4.3) als

$$\bar{\theta}(l) = \frac{t - l - s}{2t}. \tag{4.4}$$

Alle Konsumenten, die in ihrer Wahl nicht eingeschränkt sind und deren Typ $\theta < \bar{\theta}(l)$ ist, kaufen dann die proprietäre Software und alle anderen die offene. Berücksichtigt man, daß die Nachfrage auf das Einheitsintervall beschränkt ist, so erhält man die in Abbildung 4.1 dargestellte Nachfragefunktion d; der Hersteller der proprietären Software hat also für $l \in [-t - s, t - s]$ den Absatz $d(l) = (1 - z)\bar{\theta}(l)$ und den Gewinn $G(l) = l \cdot d(l)$. Diesen maximieren, wie man leicht nachprüft, Lizenzgebühren in Höhe von

$$l^* = \frac{t - s}{2}. \tag{4.5}$$

[11] Für $s \geq t$ präferieren alle Konsumenten die offene Software selbst dann, wenn der Hersteller der proprietären Software Lizenzgebühren in Höhe von 0 erhebt.

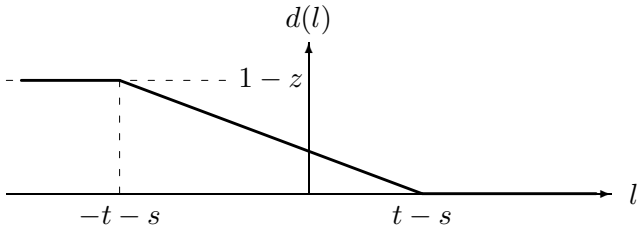


Abbildung 4.1: Nachfragefunktion im Grundmodell

Daraus resultieren der gleichgewichtige indifferente Konsument

$$\theta^* = \bar{\theta}(l^*) = \frac{t-s}{4t} \tag{4.6}$$

und der Gleichgewichtsgewinn

$$G^* = G(l^*) = \frac{(1-z)(t-s)^2}{8t}. \tag{4.7}$$

Wie zu erwarten war, schmälern wegen

$$\frac{\partial G^*}{\partial z} = \frac{-(t-s)^2}{8t} < 0 \quad \text{und} \quad \frac{\partial G^*}{\partial s} = \frac{-(1-z)(t-s)}{4t} < 0 \tag{4.8}$$

beide Instrumente den Gewinn des Herstellers der proprietären Software.

Des weiteren ergibt sich im Gleichgewicht eine Konsumentenrente in Höhe von

$$K^* = (1-z)\left(\int_0^{\theta^*} u_P(\theta, l^*)\, d\theta + \int_{\theta^*}^1 u_O(\theta)\, d\theta\right) + z\int_0^1 u_O(\theta)\, d\theta. \tag{4.9}$$

Dabei erfaßt das erste Integral die Käufer der proprietären Software, das zweite die freiwilligen und das dritte die zum Erwerb gezwungenen Käufer der offenen Software. Einfache Umformungen erbringen

$$K^* = b - \frac{t}{2} + s + \frac{(1-z)(t-s)^2}{16t}. \tag{4.10}$$

Die erzwungene Nutzung offener Software verringert, wie

$$\frac{\partial K^*}{\partial z} = \frac{-(t-s)^2}{16t} < 0 \tag{4.11}$$

zeigt, die Konsumentenrente, die Schaffung eines Zusatznutzens erhöht sie hingegen wegen

$$\frac{\partial K^*}{\partial s} = \frac{(7+z)t + (1-z)s}{8t} > 0. \tag{4.12}$$

Dabei setzt sich dieser positive Nettoeffekt aus drei Komponenten zusammen: Erstens profitieren die Käufer der offenen Software ganz unmittelbar vom Zusatznutzen und zweitens haben die Käufer der proprietären Software eine geringere Lizenzgebühr l^* zu entrichten; drittens tragen aber die Konsumenten, welche sich nur aufgrund des Zusatznutzens für die offene Software entschieden haben, zusätzliche Transportkosten.[12]

Um abschließend beurteilen zu können, ob die Schaffung eines Zusatznutzens gesamtgesellschaftlich wünschenswert ist, müssen noch die dadurch entstehenden Kosten berücksichtigt werden. Hier sollen zwei extreme Fälle unterschieden werden. Zum einen sei der Zusatznutzen kostenlos, also etwa das Ergebnis eines Informationsangebots, welches im Vergleich zur Marktgröße keinen nennenswerten Aufwand erfordert. Zum anderen kann der Zusatznutzen aber auch eine direkte Subvention der Konsumenten darstellen; die Kosten sind dann

$$T^* = ((1-z)(1-\theta^*) + z)\, s \tag{4.13}$$

mit

$$\frac{\partial T^*}{\partial s} = (1-z)\frac{3t+2s}{4t} + z. \tag{4.14}$$

Nachfolgend wird vereinfachend angenommen, daß der Staat diese Kosten über eine Kopfsteuer vollständig auf die Konsumenten abwälzt.

Neben einer Fallunterscheidung bezüglich der Kosten ist es des weiteren sinnvoll, auch im Hinblick auf die Nationalität des Herstellers der proprietären Software zu unterscheiden. Handelt es sich nämlich um ein inländisches Unternehmen, so ist sein Gewinn zweifelsfrei Bestandteil der für die staatliche Entscheidung relevanten Wohlfahrt; handelt es sich hingegen um ein ausländisches Unternehmen, so mag sein Gewinn irrelevant sein. Insgesamt ergeben sich also vier Fälle, für welche die jeweils relevante Wohlfahrt in der folgenden Matrix dargestellt ist:

[12] Auch wenn die Bezeichnung "Transportkosten" nur für die räumliche Interpretation horizontaler Produktdifferenzierung wirklich trifft, wird sie nachfolgend auch für die Kosten verwendet, welche Konsumenten dadurch entstehen, daß sie nicht ihr Idealprodukt erhalten, sondern auf eines der beiden Angebote an den Randpunkten des Einheitsintervalls zurückgreifen müssen.

	Inländisches Unternehmen	Ausländisches Unternehmen
Kosten 0	$G^* + K^*$	K^*
Kosten T^*	$G^* + K^* - T^*$	$K^* - T^*$

Differenziert man nun jeweils nach s und untersucht das Vorzeichen, so erhält man:

	Inländisches Unternehmen	Ausländisches Unternehmen
Kosten 0	$\frac{\partial G^*}{\partial s} + \frac{\partial K^*}{\partial s} > 0$	$\frac{\partial K^*}{\partial s} > 0$
Kosten T^*	$\frac{\partial G^*}{\partial s} + \frac{\partial K^*}{\partial s} - \frac{\partial T^*}{\partial s} < 0$	$\frac{\partial K^*}{\partial s} - \frac{\partial T^*}{\partial s} > 0 \iff s < t/3$

Es zeigt sich also, daß die Schaffung eines Zusatznutzens immer wünschenswert ist, wenn die Kosten dafür 0 betragen. Vielfältiger ist hingegen der Fall von Kosten in Höhe von T^*. Hier gilt:

Satz 4.1 *Im Grundmodell bewirkt eine per Kopfsteuer finanzierte Subvention der Nutzung offener Software in Höhe von* $s > 0$

- *eine Erhöhung der Konsumentenrente* K^* *dann und nur dann, wenn die Bedingung* $s < t/3$ *erfüllt ist,*
- *sowie eine Verringerung der Summe von Konsumentenrente und Unternehmensgewinn* $K^* + G^*$.

Ist also der Hersteller der proprietären Software ein inländisches Unternehmen, so haben Subventionen einen negativen Einfluß auf die Wohlfahrt. Die Ursache hierfür liegt darin, daß in diesem Fall sowohl die Subventionen als auch die Lizenzgebühren wohlfahrtsneutrale Transfers darstellen und somit das einzige Ziel staatlichen Handelns die Verringerung der Transportkosten sein kann. Diesem Ziel wird allerdings die Förderung offener Software nicht gerecht. In Abwesenheit aller staatlichen Eingriffe hat der indifferente Konsument im Gleichgewicht gemäß Gleichung (4.6) den Typ 1/4. Es kommt also allein unter dem Aspekt minimaler Transportkosten bereits ohne staatliche Eingriffe zu einer Überadoption

der offenen Software, da sie als lizenzgebührenfreies Angebot auch solche Konsumenten für sich gewinnt, deren Transportkosten beim Kauf der proprietären Software geringer gewesen wären. Die Schaffung eines zusätzlichen Anreizes zur Adoption der offenen Software verschärft dieses Problem noch.

Ist hingegen der Hersteller der proprietären Software ein ausländisches Unternehmen, so ist die Subventionierung der Nutzung offener Software bis zu einem gewissen Grade sinnvoll, wenn die Gewinne von Ausländern aus politischen Gründen ignoriert werden. Allerdings muß der Eingriff auch dann maßvoll sein. Dieses ist dadurch zu erklären, daß die Subventionen in diesem Fall zwei gegenläufige Effekte bewirken. Zum einen profitieren alle Käufer der proprietären Software von sinkenden Lizenzgebühren; dieser Effekt wird offensichtlich mit steigenden Subventionen immer schwächer, da sich die Zahl der Käufer der proprietären Software verringert. Zum anderen aber haben die Nutzer der offenen Software steigende Transportkosten zu tragen; dieser Effekt wird offensichtlich immer stärker, da sukzessive Konsumenten mit einem immer niedrigeren Typ die offene Software wählen. Für $s > t/3$ dominiert der letztgenannte, negative Effekt den erstgenannten.

Die Beurteilung der erzwungenen Adoption offener Software fällt hingegen eindeutig aus. Wie die Ungleichungen (4.8) und (4.11) zeigen, reduziert diese Maßnahme sowohl den Gewinn als auch die Konsumentenrente und ist somit eine ungeeignete Form staatlicher Einflußnahme auf den betrachteten Markt. Ursächlich dafür ist die krude Art, mit der die erzwungene Adoption die Marktanteile steuert: Ohne Ansehen ihres Typs werden Konsumenten zur Adoption gezwungen, bis die gewünschte Veränderung der Marktanteile zugunsten der offenen Software hergestellt ist. Die Schaffung von Zusatznutzen ist demgegenüber selbst in ihrer Ausgestaltung als Subvention vorzuziehen, da sie dieselben Marktanteile durch eine Verschiebung des indifferenten Konsumenten und somit zu geringeren Transportkosten erreicht. Daher wird die erzwungene Adoption im folgenden nicht mehr betrachtet. Des weiteren wird die Schaffung von Zusatznutzen nachfolgend immer als kopfsteuerfinanzierte Subvention interpretiert, um die Analyse auf den wie gesehen reichhaltigeren Fall zu fokussieren.

4.2.3 Modell mit Netzwerkeffekten

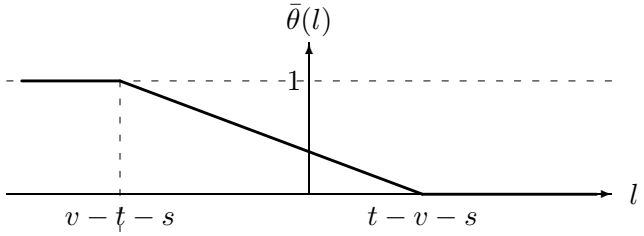
In diesem Abschnitt wird das eben vorgestellte Grundmodell um (direkte) Netzwerkeffekte erweitert, indem die Nutzenfunktionen der Konsumenten jeweils um einen zusätzlichen Summanden zu

$$u_P(\theta, l, x) = b - t \cdot \theta + v \cdot x - l \quad \text{und} \quad u_O(\theta, x) = b - t \cdot (1-\theta) + v \cdot (1-x) + s \tag{4.15}$$

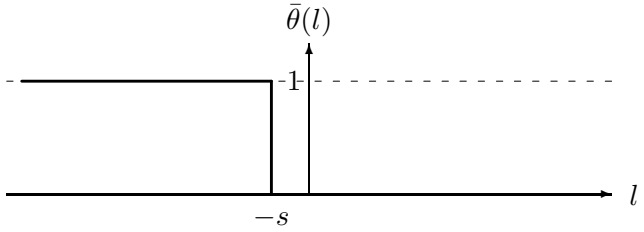
ergänzt werden. Dabei erfassen die Terme $v \cdot x$ und $v \cdot (1-x)$ den erwarteten Nutzenbeitrag der zur proprietären bzw. zur offenen Software gehörenden Konsumentennetzwerke mit $v \geq 0$ als Maß für die Stärke des Netzwerkeffekts und $x \in [0,1]$ als *erwarteter* Größe des Netzwerks der proprietären Software.

Die in dieser Spezifikation implizierte Annahme identischer Erwartungen unter allen Konsumenten ist möglich, da die nachfolgende Analyse für rationale Erwartungen durchgeführt wird und somit im Gleichgewicht die Erwartungen aller Konsumenten der eindeutigen tatsächlichen Netzwerkgröße entsprechen müssen. Bezeichnet $\hat{\theta}(l, x)$ die tatsächliche Netzwerkgröße für Lizenzgebühren in Höhe von l und eine erwartete Netzwerkgröße von x, so bedeutet dieses formal, daß eine mit der Annahme rationaler Erwartungen konsistente tatsächliche Netzwerkgröße, nachfolgend mit $\bar{\theta}(l)$ bezeichnet, die Gleichung $\hat{\theta}(l, \bar{\theta}(l)) = \bar{\theta}(l)$ erfüllt. Es wird sich zeigen, daß $\bar{\theta}(l)$ nicht immer eindeutig bestimmt ist, je nachdem, ob schwache ($v < t$) oder starke ($v \geq t$) Netzwerkeffekte vorliegen. Formal ist $\bar{\theta}$ also keine Funktion, sondern eine Korrespondenz mit der Potenzmenge von $[0,1]$ als Wertebereich. Sie gibt alle mit rationaler Erwartungsbildung konsistenten Nachfragen nach der proprietären Software in Abhängigkeit von den Lizenzgebühren an und wird daher nachfolgend Nachfragekorrespondenz genannt.

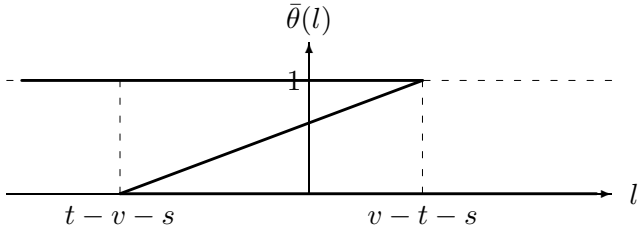
Bei der Herleitung der Nachfragekorrespondenz sind drei Fälle zu untersuchen. Erstens muß festgestellt werden, für welche Lizenzgebühren l die Randlösung $1 \in \bar{\theta}(l)$ ist. Offensichtlich liegt diese Randlösung für alle l vor, welche die Ungleichung $u_P(1, l, 1) \geq u_O(1, 1)$ erfüllen, da dann bei einer erwarteten Netzwerkgröße von $x = 1$ auch der Konsument mit dem Typ $\theta = 1$ die proprietäre



(a) für $v < t$



(b) für $v = t$



(c) für $v > t$

Abbildung 4.2: Nachfragekorrespondenz bei rationalen Erwartungen im Modell mit Netzwerkeffekten

Software der offenen vorzieht. Es ist leicht zu zeigen, daß diese Ungleichung für alle $l \leq v - t - s$ erfüllt ist. Zweitens kann analog ermittelt werden, daß für alle $l \geq t - v - s$ die Randlösung $0 \in \bar{\theta}(l)$ ist. Schließlich ist drittens noch zu untersuchen, für welche l eine innere Lösung $\theta' \in (0,1) \cap \bar{\theta}(l)$ vorliegt und wie sie gegebenenfalls lautet. Offensichtlich ist eine solche innere Lösung durch zwei Merkmale gekennzeichnet. Zum einen muß die erwartete Netzwerkgröße der tatsächlichen entsprechen, also $x = \theta'$ gelten, und zum anderen muß der Konsument vom Typ θ' zwischen beiden Angeboten indifferent sein. Für jedes l entspricht also die Gesamtheit aller inneren Lösungen der Lösungsmenge der

Gleichung

$$\begin{aligned} u_P(\theta', l, \theta') &= u_F(\theta', \theta') \\ \iff \quad b - t \cdot \theta' + v \cdot \theta' - l &= b - t \cdot (1 - \theta') + v \cdot (1 - \theta') + s. \end{aligned} \tag{4.16}$$

Ist $v < t$, so hat Gleichung (4.16) für alle $l \in (v - t - s, t - v - s)$ genau eine Lösung, nämlich

$$\theta' = \frac{(t - v) - l - s}{2(t - v)}; \tag{4.17}$$

ist $v = t$, so sind für $l = -s$ alle $\theta' \in (0, 1)$ Lösungen; ist schließlich $v > t$, so existiert für alle $l \in (t - v - s, v - t - s)$ genau eine Lösung, die wiederum durch Gleichung (4.17) gegeben ist. Weitere Lösungen gibt es nicht. Insgesamt zeigt sich also, daß die Nachfragekorrespondenz in Abhängigkeit von t und v dreierlei Gestalt annimmt, wie in Abbildung 4.2 dargestellt.

4.2.3.1 Schwache Netzwerkeffekte

Ein Vergleich der Nachfragekorrespondenz für im Vergleich zur horizontalen Produktdifferenzierung schwache Netzwerkeffekte ($v < t$) (vgl. Abbildung 4.2 (a)) mit der Nachfragefunktion im Grundmodell (vgl. Abbildung 4.1) zeigt, daß in diesem Fall beide Modelle qualitativ identisch sind. Schwache Netzwerkeffekte erhöhen lediglich die Wettbewerbsintensität, da sie die effektive Präferenzintensität von t auf $t - v > 0$ reduzieren.[13] Grundsätzlich gelten aber alle in Abschnitt 4.2.2 getroffenen Aussagen analog.

4.2.3.2 Starke Netzwerkeffekte

Starke Netzwerkeffekte ($v \geq t$) führen hingegen zu einem fundamental anderen Marktgeschehen, in dem die Erwartungen bezüglich der Netzwerkgröße von entscheidender Bedeutung sind. Sehr anschaulich illustriert dieses Abbildung 4.2 (c). Für jede Lizenzgebühr $l \in [t - v - s, v - t - s]$ ist die mit der Annahme rationaler Erwartungen konsistente Nachfrage nach der proprietären Software nicht

[13] Die Abweichung in der maximalen Nachfrage ist allein auf das Fehlen der erzwungenen Adoption im Modell mit Netzwerkeffekten zurückzuführen.

eindeutig bestimmt. Erwarten etwa alle Konsumenten, daß sich die proprietäre Software durchsetzen wird, so kaufen sie diese eben auch und ihre Erwartungen werden erfüllt. Ganz analog kann sich aber auch die Erwartung eines von der offenen Software dominierten Marktes erfüllen. Weiterhin ist die Erwartung eines geteilten Marktes rational.

Offensichtlich ist das Maximierungskalkül des Herstellers der proprietären Software ganz entscheidend davon abhängig, welche Nachfrage sich für eine bestimmte Lizenzgebühr tatsächlich einstellt. Im weiteren sollen diesbezüglich zwei polare Regimes unterschieden werden. Im sogenannten offenen Regime sei die tatsächliche Nachfrage $d_u(l) = \min\{\bar{\theta}(l)\}$, im proprietären hingegen $d_o(l) = \max\{\bar{\theta}(l)\}$. Es gilt nun:

Satz 4.2 *Im Modell mit starken Netzwerkeffekten ($v \geq t$) bewirkt eine per Kopfsteuer finanzierte Subvention der Nutzung offener Software in Höhe von $s > 0$*

- *im offenen Regime nichts*
- *und im proprietären Regime eine Erhöhung der Konsumentenrente.*

Denn im offenen Regime kann, wie Abbildung 4.2 (b-c) zeigt, der Hersteller der proprietären Software selbst für Lizenzgebühren in Höhe von 0 keine Nachfrage auf sich ziehen. Sein Marktanteil im Gleichgewicht ist daher 0. Die offene Software wird hingegen von allen Konsumenten verwendet und bedarf somit keinerlei Förderung durch die öffentliche Hand; insbesondere ist ihre per Kopfsteuer finanzierte Subventionierung in diesem Fall vollkommen ohne Wirkung. Im proprietären Regime wird der Hersteller hingegen im Gleichgewicht für $s \leq v - t$ Lizenzgebühren in Höhe von $l^* = v - t - s$ erheben und den gesamten Markt bedienen. Die Subventionierung offener Software hat dann zunächst lediglich den Effekt, eine Verringerung der Lizenzgebühren zu erzwingen und so Renten vom Hersteller zu den Konsumenten zu verschieben. Ist der Hersteller der proprietären Software ein inländisches Unternehmen, hat dieses allerdings keine

Auswirkung auf die Wohlfahrt; handelt es sich hingegen um ein ausländisches Unternehmen, so mag die Verlagerung von Renten politisch gewünscht sein. Für $s > v - t$ zieht sich der Hersteller der proprietären Software überdies auch im für ihn grundsätzlich vorteilhaften Regime aus dem Markt zurück; es gelten dann die Aussagen zum offenen Regime. Insgesamt geben Netzwerkeffekte also gegenüber dem Grundmodell kein zusätzliches Motiv für staatliches Handeln.

Das Modell zeigt aber, welch zentrale strategische Bedeutung der Beeinflussung der Erwartungen der Konsumenten bezüglich des Markterfolgs zukommt. Vor diesem Hintergrund sind auch die Vorwürfe einiger Verfechter offener Software zu sehen, Microsoft bediene sich im Wettbewerb unredlicher FUD-Taktiken. Das von Raymond (1996) herausgegebene "New Hacker's Dictionary" definiert FUD wie folgt:

> Defined by Gene Amdahl after he left IBM to found his own company: "FUD is the fear, uncertainty, and doubt that IBM sales people instill in the minds of potential customers who might be considering Amdahl products." The idea, of course, was to persuade them to go with safe IBM gear rather than with competitors' equipment. [...] After 1990 the term FUD was associated increasingly frequently with Microsoft, and has become generalized to refer to any kind of disinformation used as a competitive weapon.

Gelingt es dem Hersteller einer proprietären Software, die bei starken Netzwerkeffekten im Wettbewerb mit einer gleichwertigen offenen Software steht, bei den Konsumenten mittels FUD-Taktiken Zweifel an der konkurrierenden Software zu wecken, so kann er am Markt erfolgreich agieren. Erwarten die Konsumenten hingegen, daß sich die offene Software durchsetzen wird, so wird es für den Hersteller äußerst schwierig, einen akzeptablen Gewinn zu erzielen. Gesamtgesellschaftlich ist der Einsatz von FUD-Taktiken allerdings nur dann problematisch, wenn das proprietäre Regime unter Wohlfahrtsaspekten hinter dem offenen zurückbleibt. In diesem Modell ist das aus Symmetriegründen nicht

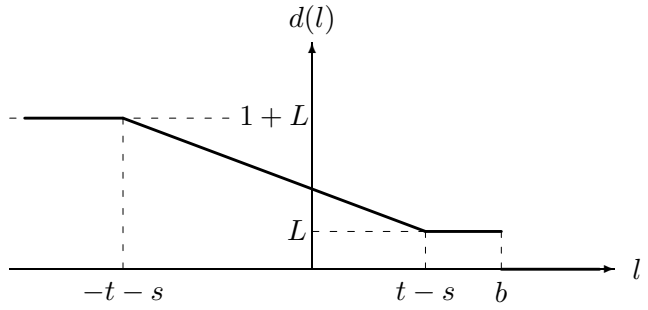


Abbildung 4.3: Nachfragefunktion im Modell mit Wechselkosten und Lock-In

der Fall; stiftet aber die offene Software etwa einen größeren Grundnutzen, so mindern FUD-Taktiken die Wohlfahrt.

4.2.4 Modell mit Wechselkosten und Lock-In

In diesem Abschnitt wird das in Abschnitt 4.2.2 vorgestellte Grundmodell um Wechselkosten erweitert. Diese werden allerdings nur implizit modelliert, indem zusätzlich zu den auf dem Einheitsintervall gleichverteilten weitere Konsumenten mit der Masse $L > 0$ eingeführt werden, die sich in einer nicht explizit modellierten Vergangenheit für eine ältere Version der proprietären Software entschieden haben und nun aufgrund prohibitiv hoher Wechselkosten nicht zur offenen Software wechseln können. Stattdessen stehen sie lediglich vor der Wahl, auf die aktuelle Version der proprietären Software umzusteigen, was einen Nutzen in Höhe von $b-l$ stiften soll, oder bei einem Nutzen in Höhe von 0 den Status Quo beizubehalten. Die Nachfrage dieser an den Hersteller der proprietären Software gebundenen Konsumenten beträgt also L für $l \leq b$ und 0 für $l > b$. Da die Nachfrage der ungebundenen Konsumenten der im Grundmodell entspricht und somit für $l \in [-t-s, t-s]$ wiederum durch Gleichung (4.4) beschrieben wird, ergibt sich die in Abbildung 4.3 dargestellte Nachfragefunktion d.

Offensichtlich gibt diese Nachfragefunktion einem seinen Gewinn $G(l) = l \cdot d(l)$ maximierenden Hersteller grundsätzlich zwei Optionen. Zum einen kann er zu Lizenzgebühren in Höhe von $l_L^* = b$ ausschließlich die an ihn gebundenen Konsumenten bedienen und so den Gewinn $G_L^* = bL$ erzielen. Da aber das Modell

das Problem der Abwägung zwischen der Gewinnung von Neukunden und der Wertabschöpfung von Altkunden erfassen soll, welches ja, wie in Abschnitt 4.2.1 dargestellt, die Preisfindung bei Wechselkosten entscheidend prägt, ist die Attraktivität dieser Option zu begrenzen; dazu wird im weiteren $L < 1/2$ und $b = t$ angenommen.[14]

Zum anderen kann der Hersteller sowohl die gebundenen als auch einen Teil der ungebundenen Konsumenten bedienen, indem er Lizenzgebühren in Höhe von $l \in (-t - s, t - s)$ erhebt; die Nachfrage beträgt dann $d(l) = L + (t - s - l)/2t$. Lizenzgebühren, welche den Gewinn $G(l) = l \cdot d(l)$ zumindest lokal maximieren, existieren in diesem Intervall nur für $s < s'(L) = t - 2tL$; sie genügen dann der notwendigen Bedingung $\partial G/\partial l = (t - s - l)/2t + L - l/2t = 0$ und betragen

$$l^*_{L+1} = \frac{t - s}{2} + tL \quad \text{mit} \quad G^*_{L+1} = G(l^*_{L+1}) = \frac{(t - s + 2tL)^2}{8t}. \tag{4.18}$$

Für $s \geq s'(L)$ existiert hingegen kein lokales Gewinnmaximum, in dem der Hersteller auch Teile der ungebundenen Konsumenten bedient, da dann die Subventionen für die offene Software zu hoch sind. Das impliziert, daß der Hersteller in diesem Fall die Lizenzgebühren l^*_L setzt und nur an die gebundenen Kunden verkauft.

Für $s < s'(L)$ ergibt sich das globale Gewinnmaximum hingegen aus dem Vergleich von G^*_L und G^*_{L+1}. Einige leichte Umformungen erbringen

$$G^*_{L+1} \geq G^*_L \iff s \leq \bar{s}(L) = t \cdot \left(1 + 2L - \sqrt{8L}\right). \tag{4.19}$$

Man beachte, daß die Schranke $\bar{s}(L)$, bei deren Überschreitung der Hersteller der proprietären Software als Ergebnis des obigen Gewinnvergleichs die Lizenzgebühren l^*_L setzt, niedriger ist als die Schranke $s'(L)$, ab der im Intervall $(-t - s, t - s)$ kein lokales Maximum mehr existiert; $s'(L)$ ist also für die Festlegung der optimalen Lizenzgebühren nicht relevant.

[14] Die Begrenzung des Grundnutzens führt dabei nicht dazu, daß einige Konsumenten keines der beiden Angebote verwenden, da die offene Software für $b = t$ selbst dem Konsumenten mit $\theta = 0$ einen positiven Nutzen stiftet.

Insgesamt müssen also im Gleichgewicht bezüglich der Lizenzgebühr l^*, dem Gewinn G^* und dem Marktanteil der proprietären Software bei den ungebundenen Konsumenten θ^* zwei Fälle unterschieden werden. Für $s \leq \bar{s}(L)$ ist[15]

$$l^* = \frac{t-s}{2} + tL, \quad G^* = \frac{(t-s+2tL)^2}{8t} \text{ und } \quad \theta^* = \frac{t-s-2tL}{4t}; \tag{4.20}$$

für $s > \bar{s}(L)$ beträgt hingegen

$$l^* = t, \quad G^* = tL \text{ und } \quad \theta^* = 0. \tag{4.21}$$

Des weiteren ergibt sich im Gleichgewicht eine Konsumentenrente in Höhe von

$$K^* = \int_0^{\theta^*} u_P(\theta, l^*)\, d\theta + L \cdot (t - l^*) + \int_{\theta^*}^1 u_O(\theta)\, d\theta - (1-\theta^*)s. \tag{4.22}$$

Dabei erfassen der erste und der zweite Summand die ungebundenen bzw. gebundenen Käufer der proprietären Software, der dritte Summand die Nutzer der offenen Software und der vierte die Kosten für die Subvention.[16] Für $s \leq \bar{s}(L)$ erbringen einfache Umformungen

$$K^*(s) = \frac{L \cdot (t - 2tL + s) + t}{2} + \frac{(t-s-2tL)(t+3s-2tL)}{16t}, \tag{4.23}$$

für $s > \bar{s}(L)$ hingegen

$$K^*(s) = \frac{t}{2}. \tag{4.24}$$

Im Grundmodell wurde gezeigt, daß eine über eine Kopfsteuer finanzierte maßvolle Subventionierung ($s < t/3$) offener Software die Konsumentenrente erhöht. Nachfolgend wird analysiert, wie dieses Ergebnis von der Existenz durch Wechselkosten gebundener Konsumenten beeinflußt wird.

Grundsätzlich erhöhen auch in dieser Modellvariante maßvolle Subventionen die Konsumentenrente, da sie den Hersteller der proprietären Software wiederum zu Preissenkungen bewegen. Blendet man für einen Moment den Umstand aus, daß

[15] Dabei wird angenommen, daß der Hersteller der proprietären Software bei gleichem Gewinn einen höheren Absatz präferiert.

[16] Anders als im Grundmodell wird hier der Zusatznutzen s immer als über eine Kopfsteuer finanzierte Subvention gesehen. Es ist daher sinnvoll, das Kopfsteueraufkommen unmittelbar in die Konsumentenrente einzubeziehen.

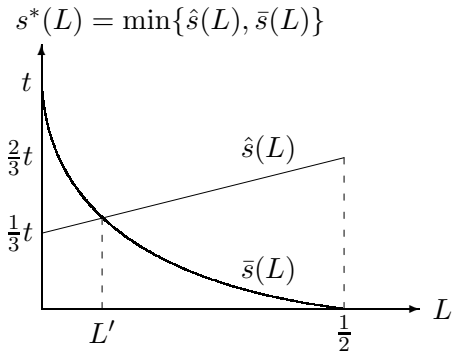


Abbildung 4.4: Obere Schranke für Subventionen im Modell mit Wechselkosten

der Hersteller für $s > \bar{s}(L)$ nur noch die gebundenen Konsumenten bedient, so ist die Konsumentenrente durch Gleichung (4.23) gegeben. Partielles Ableiten nach s erbringt

$$\frac{\partial K^*}{\partial s} = \frac{t + 2tL - 3s}{8t}. \tag{4.25}$$

Offensichtlich ist dieser Term für alle $s \leq \hat{s}(L) = (t + 2tL)/3$ positiv; Subventionen bis zu dieser oberen Schranke erhöhen also aufgrund ihrer preissenkenden Wirkung die Konsumentenrente, solange der Hersteller auch ungebundene Konsumenten bedient. Nun wird aber der Hersteller wie gezeigt für $s > \bar{s}(L)$ seine Lizenzgebühren schlagartig auf t erhöhen. Die Konsumentenrente verringert sich dadurch nach Gleichung (4.24) auf $t/2$ und liegt dann, wie man leicht nachprüft, sogar unterhalb von $K^*(0)$. Folglich gilt:

Satz 4.3 *Im Modell mit Wechselkosten und Lock-In erhöhen per Kopfsteuer finanzierte Subvention der Nutzung offener Software in Höhe von $s > 0$ die Konsumentenrente dann und nur dann, wenn gilt:*

$$s \leq s^*(L) = \min\{\hat{s}(L), \bar{s}(L)\}.$$

Der Verlauf der oberen Schranke $s^*(L)$ ist in Abbildung 4.4 dargestellt; $L' \approx 0,07$ ergibt sich dabei als Lösung von $\hat{s}(L) = \bar{s}(L)$.

Offensichtlich hat die Anzahl gebundener Konsumenten L mitunter einen erheblichen Einfluß darauf, bis zu welcher Höhe $s^*(L)$ Subventionen die Konsu-

mentenrente erhöhen. Für $L \leq L'$ steigt dabei $s^*(L)$, da immer mehr Konsumenten von den Preissenkungen profitieren und so die negative, transportkostenerhöhende Wirkung der Subventionen wirksamer kompensiert wird. Für $L \geq L'$ fällt $s^*(L)$ hingegen mit wachsendem L, da für den Hersteller die Option, nur noch die gebundenen Konsumenten zu bedienen, zunehmend attraktiver wird. Sollte also die öffentliche Hand das Ziel verfolgen, über eine Subvention offener Software die Konsumentenrente zu steigern, muß sie Wechselkosten und gebundene Konsumenten unbedingt berücksichtigen. Anderenfalls besteht die Gefahr, daß sogar kleinste Subventionen einen kontraproduktiven Effekt haben.

4.2.5 Modell mit Systemwettbewerb

In diesem Abschnitt wird das Grundmodell um den in Abschnitt 4.2.1 beschriebenen Systemwettbewerb erweitert. Obgleich ein derartiger Wettbewerb vielfältige Formen annehmen kann, ist er grundsätzlich immer dadurch gekennzeichnet, daß die Konsumenten mehrere komplementäre Komponenten erwerben und zu einem Gesamtsystem kombinieren müssen. Um diesen Aspekt im vorliegenden Fall abzubilden, wird angenommen, die offene und die proprietäre Software aus dem Grundmodell bildeten als Betriebssysteme den unverzichtbaren Kern eines Softwaresystems, welches darüberhinaus aus verschiedenen, gemäß individueller Präferenzen ausgewählten Anwendungen bestehe. Dabei seien die beiden konkurrierenden Betriebssysteme inkompatibel, d. h. Anwendungen, welche für das eine Betriebssystem entwickelt wurden, können auf dem anderen nicht eingesetzt werden. Ein Beispiel für eine derartige Konstellation ist der Wettbewerb zwischen den Betriebssystemen Windows und Linux; das Spektrum möglicher Anwendungen reicht dann von Textverarbeitungen und Tabellenkalkulationen über Computerspiele bis hin zu E-Mail- und Web-Server-Software, wobei der Schwerpunkt davon abhängt, welches Segment des Marktes für Betriebssysteme betrachtet wird.

4.2.5.1 Der Markt für Anwendungen

Zunächst soll der Markt für Anwendungen beschrieben werden: In ihm existiere ein Kontinuum von Entwicklern, die jeweils als Monopolist eine spezifische Anwendung entwickeln und anbieten können. Für jede dieser Anwendungen haben die Konsumenten unabhängig von ihrem Typ θ, welcher ja lediglich die Präferenzen im Markt für Betriebssysteme charakterisiert, eine zufällige Zahlungsbereitschaft ω; diese sei auf dem Intervall $[0, z]$ mit $z > 0$ als maximaler Zahlungsbereitschaft gleichverteilt. Erwartet also ein Anwendungsentwickler, daß $x \in [0, 1]$ Konsumenten das proprietäre Betriebssystem erwerben, sieht er sich der Nachfragefunktion $d_P(p_P) = (1-p_P/z)x$ mit p_P als Preis seiner Anwendung für das proprietäre Betriebssystem gegenüber. Er maximiert dann seinen Umsatz $p_P \cdot d_P(p_P)$,[17] indem er den Anwendungspreis auf $p_P^* = z/2$ setzt und erzielt so den maximalen Umsatz $(z/4)x$; analog erhält man für Anwendungen, die für das offene Betriebssystem entwickelt werden, ebenfalls den optimalen Preis $p_O^* = z/2$ und den maximalen Umsatz $(z/4)(1-x)$.

Die Frage, ob und für welche Betriebssysteme ein Anwendungsentwickler überhaupt anbietet, hängt natürlich zusätzlich von seinen fixen Entwicklungskosten ab. Hier wird angenommen, daß jeder Entwickler durch Fixkosten $f \geq 0$ charakterisiert sei, welche sowohl bei der Entwicklung für das proprietäre als auch für das offene Betriebssystem anfallen. Des weiteren soll es keine Verbundvorteile geben; wird also eine Anwendung für beide Betriebssysteme entwickelt, so entstehen dadurch Kosten in Höhe von $2f$. Zudem fördere die öffentliche Hand die Entwicklung von Anwendungen für das offene Betriebssystem mit Subventionen in Höhe von $s_a \geq 0$. Der Gewinn eines Entwicklers mit Fixkosten f und Erwartungen x bei Entwicklung für das proprietäre und das offene Betriebssystem ist also gegeben durch

$$g_P(f, x) = (z/4)x - f \quad \text{bzw.} \quad g_O(f, x) = (z/4)(1-x) + s_a - f. \tag{4.26}$$

Selbstverständlich werden nur solche Anwendungen entwickelt, mit denen ein

[17] Wiederum werden Grenzkosten in Höhe von 0 angenommen.

positiver Gewinn erzielt werden kann. Es wird nun angenommen, die Masse der Entwickler mit Entwicklungskosten kleiner oder gleich f betrage ebenfalls f; besteht also beispielsweise die Erwartung, mit einer Anwendung für ein bestimmtes Betriebssystem lasse sich ein Umsatz in Höhe von zwei Geldeinheiten erzielen, so werden insgesamt auch zwei Anwendungen entwickelt. Und allgemein ergibt sich das Anwendungsangebot für das proprietäre und das offene Betriebssystem bei Erwartungen x als Lösung von $g_P(f,x) = 0$ bzw. $g_O(f,x) = 0$. Es beträgt

$$f_P = (z/4)x \quad \text{und} \quad f_O = (z/4)(1-x) + s_a. \tag{4.27}$$

4.2.5.2 Der Markt für Betriebssysteme

Bei inkompatiblen Betriebssystemen ist das Anwendungsangebot auch für die Kaufentscheidung in diesem Markt relevant. Da die Zahlungsbereitschaft jedes Konsumenten für jede Anwendung durch eine auf dem Intervall $[0,z]$ gleichverteilte Zufallsvariable beschrieben wird, stiftet eine Anwendung mit einem Preis in Höhe von $z/2$ einem Konsumenten einen erwarteten Nutzen in Höhe von

$$\int_{z/2}^{z} \frac{1}{z}(\omega - z/2)\, d\omega = z/8. \tag{4.28}$$

Dabei umfassen die Integrationsgrenzen all jene Zahlungsbereitschaften ω, für die eine zum Preis $z/2$ angebotene Anwendung tatsächlich erworben wird und den Nettonutzen $\omega - z/2$ stiftet.

Die Entscheidung, eine Anwendung zu entwickeln, hat einen langfristigeren Charakter als eine Konsumentscheidung und geht dieser somit zeitlich voran; das Angebot an Anwendungen sei den Konsumenten bei ihrer Kaufentscheidung im Markt für Betriebssysteme daher bekannt. Die Nutzenfunktionen der Konsumenten die beiden Betriebssysteme betreffend sind folglich

$$u_P(\theta, l, f_P) = b - t \cdot \theta + (z/8) \cdot f_P - l \tag{4.29}$$

und

$$u_O(\theta, f_O) = b - t \cdot (1-\theta) + (z/8) \cdot f_O + s_k. \tag{4.30}$$

Dabei erfassen die Terme $(z/8) \cdot f_P$ und $(z/8) \cdot f_O$ den Nutzenbeitrag des für das proprietäre bzw. das offene Betriebssystem vorhandenen Angebots an Anwendungen und s_k die an die Konsumenten für die Nutzung des offenen Betriebssystems gezahlten Subventionen.

Einsetzen des durch die Gleichungen (4.27) gegebenen Anwendungsangebots in die Nutzenfunktionen (4.29) und (4.30) erbringt

$$u_P(\theta, l, x) = b - t \cdot \theta + (z^2/32)x - l \tag{4.31}$$

und

$$u_O(\theta, x) = b - t \cdot (1 - \theta) + (z^2/32)(1 - x) + (z/8)s_a + s_k. \tag{4.32}$$

4.2.5.3 Analyse und Ergebnisse

Es ist leicht zu sehen, daß diese Nutzenfunktionen und die in Gleichung (4.15) für das Modell mit Netzwerkeffekten angegebenen qualitativ identisch sind. Der Markt für Anwendungen induziert also in dem für Betriebssysteme indirekte Netzwerkeffekte der Stärke $v = z^2/32$; des weiteren betragen die effektiven Subventionen für die Nutzer des freien Betriebssystems $s = (z/8)s_a + s_k$. Unterstellt man wiederum, daß im Gleichgewicht der erwartete Marktanteil x der freien Software dem tatsächlichen θ^* entspricht, läßt sich dieses folglich wie in Abschnitt 4.2.3 ermitteln. Des weiteren ergibt sich aus den Gleichungen (4.27) und $x = \theta^*$ das gleichgewichtige Anwendungsangebot

$$f_P^* = (z/4)\theta^* \quad \text{und} \quad f_O^* = (z/4)(1 - \theta^*) + s_a. \tag{4.33}$$

Während die Herleitung des Gleichgewichts des in diesem Abschnitt präsentierten Modells keiner erneuten Darstellung bedarf, wird die Analyse des Einflusses von Subventionen auf die Wohlfahrt aus zwei Gründen weiter vertieft. Erstens sind nun auch die auf dem Markt für Anwendungen erzielten Produzentenrenten zu berücksichtigen und zweitens existiert in diesem Modell mit der Subventionierung der Anwendungsentwickler ein neues, angebotsseitiges Instrument, welches im Modell mit direkten Netzwerkeffekten nicht darstellbar war.

Hier offenbart sich die Schwäche zahlreicher Arbeiten zu Netzwerkeffekten, zwischen direkten und indirekten Netzwerkeffekten zwar verbal zu unterscheiden, sie jedoch formal vollkommen identisch zu behandeln.

Die gesamtgesellschaftliche Wohlfahrt beinhaltet in dieser Modellvariante drei Komponenten, die Konsumentenrente K^*, den Gewinn des Herstellers des proprietären Betriebssystems G^* sowie den Gesamtgewinn der Anwendungsentwickler A^*. Sie beträgt für eine gleichgewichtige Lizenzgebühr in Höhe von l^*, einen gleichgewichtigen Marktanteil der proprietären Software in Höhe von θ^*, und gleichgewichtige Anwendungsangebote in Höhe von f_P^* und f_O^*

$$\begin{aligned} W^* &= \underbrace{\int_0^{\theta^*} u_P(\theta, l^*, \theta^*)\, d\theta + \int_{\theta^*}^1 u_O(\theta, \theta^*) - s_k\, d\theta}_{K^*} + \underbrace{l^*\theta^*}_{G^*} \\ &\quad + \underbrace{\int_0^{f_P^*} g_P(f, \theta^*)\, df + \int_0^{f_O^*} g_O(f, \theta^*) - s_a\, df}_{A^*}. \end{aligned} \tag{4.34}$$

Da l^*, f_P^* und f_O^* allesamt Funktionen von θ^* sind, ergibt sich nach einigen Umformungen

$$\begin{aligned} W^* &= \left[z^2/8 - t\right]\theta^{*2} + \left[-z^2/8 - (z/8)s_a + t\right]\theta^* \\ &\quad + \left[b + z^2/16 + (z/8)s_a - (1/2)s_a^2 - t/2\right]. \end{aligned} \tag{4.35}$$

Zunächst wird der Fall starker Netzwerkeffekte ($z^2/32 \geq t$) untersucht. Gemäß der Analyse in Abschnitt 4.2.3 können für diesen Fall zwei alternative Regimes unterschieden werden: ein offenes, in welchem entsprechende Erwartungen der Konsumenten die vollständige Verdrängung des proprietären Systems bewirken, und – mit entgegengesetztem Ergebnis – ein proprietäres Regime.

Im offenen Regime beträgt $\theta^* = 0$ und somit nach Gleichung (4.35) die Wohlfahrt

$$W_{OR}^* = b + z^2/16 + (z/8)s_a - (1/2)s_a^2 - t/2. \tag{4.36}$$

Man erkennt, daß die Subventionierung der Konsumenten keinen Einfluß auf die Wohlfahrt hat, da sie in einem Zustand, in dem ohnehin schon alle Konsumenten das offene System wählen, keine allokative Wirkung entfaltet. Ein anderes Bild ergibt sich hinsichtlich der Subventionierung der Anwendungsentwickler. Hier gilt:

Satz 4.4 *Im Modell mit Systemwettbewerb existiert für* $z^2/32 \geq t$ *ein Gleichgewicht, in dem alle Konsumenten das offene Betriebssystem wählen. In diesem Gleichgewicht erhöht eine per Kopfsteuer finanzierte Subvention der Entwicklung von Anwendungen für das offene Betriebssystem in Höhe von* $s_a > 0$ *die Wohlfahrt dann und nur dann, wenn gilt:*

$$\partial W^*_{OR}/\partial s_a = z/8 - s_a > 0 \iff s_a < z/8.$$

Die Ursache hierfür liegt allerdings nicht in einem Spezifikum offener Software begründet, sondern in einer Unvollkommenheit des Marktes für Anwendungssoftware. Diese besteht im gegebenen Modell darin, daß die jeweils als Monopolist agierenden Anwendungsentwickler sich die Zahlungsbereitschaft ihrer Kunden nicht vollständig aneignen können. Daraus resultiert ein unter Wohlfahrtsaspekten zu geringes Anwendungsangebot, welches aber Subventionen zu erhöhen vermögen.

Im proprietären Regime[18] beträgt hingegen $\theta^* = 1$ und somit nach Gleichung (4.35) die Wohlfahrt

$$W^*_{PR} = b + z^2/16 - (1/2)s_a^2 - t/2. \tag{4.37}$$

Wiederum hat die Subventionierung der Konsumenten keine Wirkung, wohingegen die der Anwendungsentwickler wegen $\partial W^*_{PR}/\partial s_a = -s_a \leq 0$ die Wohlfahrt reduziert. Die Ursache hierfür liegt darin, daß im proprietären Regime kein Konsument Anwendungen für das offene System erwirbt und deren Entwicklung somit Kosten verursacht, ohne Nutzen zu stiften. Des weiteren sei darauf hingewiesen, daß die Subventionierung der Entwicklung von Anwendungen für das proprietäre System in diesem Regime aus Symmetriegründen denselben positiven Wohlfahrtseffekt hat wie für den Fall des offenen Regimes bereits beschrieben.

Nun wird der Fall schwacher Netzwerkeffekte ($z^2/32 < t$) untersucht. In Abschnitt 4.2.3 wurde gezeigt, daß dieser Fall prinzipiell dem Grundmodell ent-

[18] In Abschnitt 4.2.3 wurde gezeigt, daß dieses Regime nur dann existieren kann, wenn die effektiven Subventionen hinreichend klein sind ($(z/8)s_a + s_k \leq z^2/32 - t$).

spricht. Man erhält also den gleichgewichtigen Marktanteil der proprietären Software θ^*, indem man in Gleichung (4.6) die Präferenzintensität t durch die effektive Präferenzintensität $t - z^2/32$ und die Subventionen s durch $(z/8)s_a + s_k$ ersetzt; er lautet

$$\theta^* = \frac{32t - z^2 - 4zs_a - 32s_k}{128t - 4z^2}. \tag{4.38}$$

Durch Einsetzen von Gleichung (4.38) in Gleichung (4.35) erhält man des weiteren die Wohlfahrt

$$\begin{aligned} W^*_{SW} &= \left[z^2/8 - t\right] \left(\tfrac{32t - z^2 - 4zs_a - 32s_k}{128t - 4z^2}\right)^2 \\ &\quad + \left[-z^2/8 - (z/8)s_a + t\right] \left(\tfrac{32t - z^2 - 4zs_a - 32s_k}{128t - 4z^2}\right) \\ &\quad + \left[b + z^2/16 + (z/8)s_a - (1/2)s_a^2 - t/2\right]. \end{aligned} \tag{4.39}$$

Es ist schwierig, den allgemeinen Zusammenhang zwischen den Subventionen s_a und s_k und der Wohlfahrt festzustellen. Jedoch kann anhand der Vorzeichen der partiellen Ableitungen von W^*_{SW} nach s_a und s_k für $s_a = s_k = 0$ immerhin ermittelt werden, ob geringfügige Subventionen die Wohlfahrt steigern können. Es zeigt sich, daß wegen

$$\left.\frac{\partial W^*_{SW}}{\partial s_a}\right|_{s_a=0, s_k=0} = \frac{z \cdot (80t - z^2)}{32 \cdot (32t - z^2)} > 0 \tag{4.40}$$

die Subventionierung der Anwendungsentwickler sinnvoll ist. Die Ursache dafür entspricht der bereits für den Fall starker Netzwerkeffekte beschriebenen. Der Wohlfahrtseffekt der Subventionierung von Konsumenten ist hingegen überraschend. Bislang hat dieses Instrument die Wohlfahrt aufgrund seiner allein die Transportkosten erhöhenden Wirkung immer reduziert. Im Modell mit Systemwettbewerb gilt hingegen:

Satz 4.5 *Im Modell mit Systemwettbewerb existiert für $z^2/32 < t$ ein eindeutiges Gleichgewicht, in dem sich das offene und das proprietäre Betriebssystem den Markt teilen. In diesem Gleichgewicht erhöht eine hinreichend kleine per Kopfsteuer finanzierte Subvention der Nutzung offener Software in Höhe von $s_k > 0$ die Wohlfahrt dann und nur dann, wenn gilt:*

$$\left.\frac{\partial W^*_{SW}}{\partial s_k}\right|_{s_a=0, s_k=0} = \frac{z^2 - 8t}{2(32t - z^2)} > 0 \iff z^2 > 8t.$$

Die maßvolle Subventionierung der Konsumenten vermag also die Wohlfahrt zu erhöhen, wenn die maximale Zahlungsbereitschaft für Anwendungen z relativ zur Präferenzintensität t hinreichend groß ist. Zwar erhöht die Subventionierung der Konsumenten nach wie vor die Transportkosten im Markt für Betriebssysteme und mindert so die Wohlfahrt. Diese Minderung wird allerdings unter gewissen Umständen durch einen Effekt im Markt für Anwendungssoftware ausgeglichen. Es ist leicht zu zeigen, daß der in Gleichung (4.38) angegebene Marktanteil des proprietären Systems $\theta^* < 1/2$ ist. Somit übersteigt nach Gleichung (4.33) das Anwendungsangebot für das offene System dasjenige für das proprietäre. Aus Gleichung (4.26) ist ersichtlich, daß der Gewinn eines Anwendungsanbieters für das proprietäre System im gleichen Maße sinkt, wie der eines Anwendungsanbieters für das offene System steigt, wenn Konsumenten vom proprietären zum offenen System wechseln. Da es aber mehr Anwendungsanbieter für das offene System gibt, bewirkt der durch die Subventionierung induzierte Wechsel von Konsumenten zum offenen System eine Steigerung der aggregierten Gewinne der Anwendungsentwickler. Für großes z fällt diese Gewinnsteigerung besonders hoch aus; für $z^2 > 8t$ schließlich vermag sie die durch die Subventionen bewirkte Steigerung der Transportkosten zu kompensieren. Es sei aber angemerkt, daß die positive Gesamtwirkung der Subventionierung von Konsumenten in hohem Maße von der Struktur des Marktes für Anwendungssoftware abhängt. Des weiteren spielen die Spezifika offener Software nur insoweit eine Rolle, als daß das Fehlen von Lizenzgebühren die aus Sicht der Anwendungsanbieter erfreuliche Konzentration der Konsumenten auf eines der konkurrierenden Betriebssysteme begünstigt.

4.2.6 Verwandte Arbeiten

Arbeiten von Schmidt und Schnitzer (2003) sowie Comino und Manenti (2005) beinhalten Modelle, die den oben präsentierten formal ähnlich sind, aber andere Aspekte der staatlichen Förderung offener Software adressieren. Sie werden nachfolgend kurz beschrieben.

Schmidt und Schnitzer (2003) analysieren mit Hilfe eines Modells, welches dem in Abschnitt 4.2.4 vorgestellten mit Wechselkosten gleicht, die Wirkung erzwungener Nutzung offener Software durch öffentliche Einrichtungen. Sie zeigen, daß ein derartiger Zwang alle Marktteilnehmer schlechter stellt. Des weiteren untersuchen sie den Einfluß dieser Maßnahme auf die Anreize des Herstellers der proprietären Software, den Grundnutzen zu erhöhen, und kommen zu dem Ergebnis, daß diese Anreize schwinden. Die Wirkung von Zwang ist also wieder negativ. In einem informellen Teil diskutieren Schmidt und Schnitzer (2003) zudem diverse weitere Aspekte eines staatlichen Eingreifens in Softwaremärkte zugunsten offener Software. Insgesamt beurteilen sie derartige Maßnahmen als nicht empfehlenswert.

Comino und Manenti (2005) erweitern das in Abschnitt 4.2.2 behandelte Grundmodell um sogenannte *uninformierte* Konsumenten, die nicht um die Existenz offener Software wissen. Diese Konsumenten kaufen die proprietäre Software, wenn sie dadurch einen positiven Nutzen erhalten; anderenfalls verwenden sie schlicht keine Software. Comino und Manenti (2005) zeigen nun, daß die erzwungene Nutzung offener Software die Wohlfahrt erhöht, wenn hinreichend viele uninformierte Konsumenten existieren. Des weiteren erhöhen die Wohlfahrt auch alle Maßnahmen, welche die Anzahl uninformierter Konsumenten verringern und keine Kosten verursachen. Subventionen entfalten hingegen immer eine negative Wirkung. Zudem analysieren Comino und Manenti (2005) ihr Modell auch für Netzwerkeffekte, können aber leider nur numerische Ergebnisse vorweisen. Jene legen nahe, daß Maßnahmen, die ein Tipping des Marktes in ein offenes Regime bewirken, die Wohlfahrt erhöhen. Ob aber der Staat im Wettbewerb konkurrierender Softwareprodukte um den Markt den Gewinner wählen sollte, muß bezweifelt werden.

4.3 Märkte für Individualsoftware

In diesem Abschnitt wird die Rolle offener Software in Märkten für Individualsoftware analysiert. Unter Individualsoftware soll dabei solche Software verstan-

	Jahr							
Art	1997	1998	1999	2000	2001	2002	2003	2004
Massensoftware	40†	48	53	57	56	59	59	60
Auftragsentwicklungen	47	55	64	74	72	61	58	57
Eigenentwicklungen	42	45	61	72	75	74	80	88
Gesamt	128	147	178	203	203	194	196	205

†alle Angaben in Mrd. $

Tabelle 4.1: Softwareinvestitionen in den USA von 1997 bis 2004

den werden, die speziell auf die Bedürfnisse eines Nutzers zugeschnitten wird und die einen Funktionsumfang bereitstellt, der über den von Massensoftware hinausgeht. Entwickelt wird derartige Software entweder hausintern oder durch die Vergabe von Aufträgen an entsprechende Dienstleistungsunternehmen.

Da Individualsoftware öffentlich kaum in Erscheinung tritt, wird ihre Bedeutung chronisch unterschätzt. Tatsächlich kommt ihr aber ein weit höherer Rang zu als Massensoftware. Tabelle 4.1 zeigt auf der Grundlage von Angaben des Bureau of Economic Analysis (2005) für den Zeitraum von 1997 bis 2004 die Softwareinvestitionen in den USA in Massen- und Individualsoftware, wobei im letzten Fall noch zwischen Auftrags- und Eigenentwicklungen unterschieden wird; offensichtlich sind durchgängig rund 70 % aller Investitionen in die Entwicklung von Individualsoftware geflossen.

Die Bedeutung von Individualsoftware erwächst daraus, daß der Konfigurierbarkeit von Massensoftware aufgrund sehr schnell anwachsender *Komplexitätskosten* für das Testen und die Fehlerbereinigung enge Grenzen gesetzt sind. Beispielsweise existieren für eine Software, die ihren Nutzern 50 Einstellungsmöglichkeiten mit jeweils nur zwei Optionen bietet, bereits über eine Billiarde verschiedene Konfigurationen. Selbst der bescheidene Anspruch, immerhin der überwiegende Teil all dieser Konfigurationen solle fehlerfrei arbeiten, ist bereits äußerst schwer umzusetzen. So hat Microsoft, der Prototyp eines Herstellers von Massensoftware, laut Cusumano und Selby (1995) im Jahr 1995 rund 3.900 Softwareingenieure beschäftigt, die ausschließlich mit Tests und der Bearbei-

tung von Kundenwünschen betraut waren, und nur rund 1.850 echte Entwickler, welche wiederum circa 50 % ihrer Arbeitszeit mit der Bereinigung von Fehlern verbrachten. Microsoft muß also für die Beherrschung der Komplexität seiner Produkte sehr viel mehr Ressourcen aufwenden als für die Entwicklung neuer Funktionen.

4.3.1 Wichtige Aspekte der Entwicklung von Individualsoftware

Wie gesehen wird Individualsoftware häufig auf der Grundlage frei ausgehandelter Verträge als Auftragsarbeit entwickelt. Der Abschluß derartiger Verträge ist aber mitunter nicht trivial. So argumentiert Bessen (2005), es sei unmöglich, einen vollständigen Vertrag über die Entwicklung einer Individualsoftware zu verfassen, da deren einzige eindeutige Beschreibung ihr Quelltext sei. Der Vertragsgegenstand ist also ex ante allenfalls unscharf beschreibbar. Anhand eines geeigneten Modells demonstriert Bessen (2005), wie dieses Problem *unvollständiger Verträge* in eine ineffiziente Entwicklung münden kann.

Des weiteren können *asymmetrische Informationen* über die Kosten und Zahlungsbereitschaften ex post ineffiziente Verhandlungsergebnisse bewirken. Eine ausführliche Übersicht über die entsprechende Literatur, die sich unverändert auf die hier behandelte Situation übertragen läßt, geben Ausubel, Cramton und Deneckere (2002); ein zentrales Ergebnis besteht darin, daß Ex-Post-Effizienz dann und nur dann erreicht werden kann, wenn ein Vertragsabschluß mit Sicherheit wünschenswert und dies zudem Common Knowledge ist.

Ein weiteres Hindernis für die Auftragsentwicklung erwächst aus der sogenannten *Sticky Information*.[19] Den Ausgangspunkt bildet die Tatsache, daß die Entwicklung einer Individualsoftware zwei Arten von Wissen erfordert: solches um die Bedürfnisse des Nutzers und solches über technische Methoden und Verfahren zur Befriedigung dieser Bedürfnisse. Über die erstgenannte Art von Wissen, im weiteren als Anwendungswissen bezeichnet, verfügt offensichtlich in erster Linie der potentielle Nutzer; die zweitgenannte Art, das sogenannte Methoden-

[19] Von Hippel (1998) stellt die Idee der Sticky Information umfassend vor.

wissen, ist hingegen vornehmlich beim Softwareentwickler zu finden. Bei der Auftragsentwicklung werden Anwendungs- und Methodenwissen beim externen Entwickler zusammengezogen. Dieser Informationstransfer induziert aber Kosten, weil es regelmäßig schwer vermittelbares Wissen, eben Sticky Information, zu transferieren gilt. Von Hippel (1998) argumentiert nun, Sticky Information könne zu einer Verschiebung des wirtschaftlichsten Orts für die Entwicklung der Software zum Nutzer führen, wenn das Methoden- leichter als das Anwendungswissen zu übertragen ist.

Die genannten Schwierigkeiten bei der Auftragsentwicklung erklären den überraschend hohen Anteil von Eigenentwicklungen an den Softwareinvestitionen in Tabelle 4.1. Es wäre allerdings falsch zu glauben, Eigenentwicklungen müßten ohne externe Hilfe gleichsam bei Null beginnen. Vielmehr existieren in vielen Einsatzfeldern Produkte, die bestimmte Basisfunktionen bereitstellen und über eine exakt spezifizierte Schnittstelle, nachfolgend technisch ungenau als "Application Programming Interface" (API) bezeichnet, um die individuell benötigten Funktionen erweitert werden können.[20] Bessen (2005) merkt an, daß die Nutzung offener Software für Eigenentwicklungen ökonomisch der Verwendung eines kostenlosen APIs gleicht (vgl. auch Abschnitt 2.5.4).

4.3.2 Modellbeschreibung

Betrachtet wird eine Modellwelt, in der Individualsoftware aus zwei Komponenten besteht: einer Basiskomponente, die allgemein gebräuchliche, auch in Massensoftware vorhandene Funktionen umfaßt, und einer Erweiterungskomponente, die jeweils individuelle Anforderungen adressiert.[21] Beide Funktionsbereiche sind für sich genommen völlig nutzlos; die Verwendung einer Massensoftware kommt daher nicht in Frage. Formal sei jede Individualsoftware charakterisiert

[20] Die Unternehmenssoftware SAP R/3 beschreitet einen solchen Weg.

[21] Beispielsweise benötigt das Internet-Auktionshaus eBay zum einen Basisfunktionen wie eine Angebots- und eine Nutzerverwaltung, die in jeder standardisierten Webshop-Software vorhanden sind, zum anderen aber auch ganz spezifische Erweiterungsfunktionen, die das firmeneigene Auktionsmodell implementieren.

durch die Qualitätsniveaus $b \in [0,1]$ und $e \in [0,1]$ der Basis- bzw. der Erweiterungskomponente. Man beachte, daß eine in der Qualität b vorliegende Basiskomponente in jede Individualsoftware integriert werden kann, während eine in der Qualität e vorliegende Erweiterungskomponente an genau einen Nutzer gebunden ist.

Des weiteren existiere ein Kontinuum von Nutzern der Masse 1. Dabei sei jeder Nutzer durch einen Typ θ beschrieben, der auf dem Einheitsintervall gleichverteilt sei. Eine Individualsoftware der Qualität (b,e) stifte diesem Nutzer den Nutzen

$$u(\theta, b, e) = \theta \cdot (b + e). \tag{4.41}$$

Der Typ θ ist also ein Maß für die Zahlungsbereitschaft eines Nutzers.

Mit Hilfe des Modells soll untersucht werden, wie sich die Unterschiede in der Ausgestaltung der geistigen Eigentumsrechte zwischen proprietärer und offener Software auf die Effizienz der Entwicklung von Individualsoftware auswirken. Ausdrücklich ausgeklammert werden sollen hingegen all jene offensichtlichen Effekte, die aus abweichenden Kostenfunktionen resultieren. Die Kosten für die Entwicklung einer Komponente der Qualität q betragen daher $c(q) = q^2/2$, und zwar unabhängig davon, wer die Entwicklungsarbeit leistet.

4.3.3 Modellanalyse

4.3.3.1 Eigenentwicklung

Selbstverständlich verfügt jeder Nutzer θ über die Möglichkeit, die von ihm benötigte Software in Eigenregie zu entwickeln. Er maximiert dann seinen Nutzen abzüglich der von ihm selbst zu tragenden Kosten

$$U_E(\theta, b, e) = u(\theta, b, e) - c(b) - c(e) = \theta \cdot (b + e) - b^2/2 - e^2/2, \tag{4.42}$$

indem er die Qualitätsniveaus $b_E^*(\theta) = \theta$ und $e_E^*(\theta) = \theta$ wählt, und erreicht so das Nutzenniveau $U_E^*(\theta) = \theta^2$. Dieses Nutzenniveau ist nachfolgend hauptsächlich deshalb von Belang, da es als Drohpunkt während Vertragsverhandlungen bei der Auftragsentwicklung dienen kann.

Ist Eigenentwicklung die einzige Möglichkeit, die benötigte Individualsoftware zu erlangen, beträgt die Wohlfahrt

$$W_E^* = \int_0^1 U_E^*(\theta)\, d\theta = 1/3. \tag{4.43}$$

Ein Quell der Ineffizienz ist dabei die redundante Entwicklung der Basiskomponente.

4.3.3.2 Auftragsentwicklung

Es gebe nun neben den Nutzern ein Unternehmen, welches über Exklusivrechte an einer Basiskomponente der Qualität $b_A = 1$ verfüge. Woher diese Basiskomponente stammt, liegt dabei außerhalb des Modellrahmens; möglicherweise ist sie das Nebenprodukt einer Betätigung des Unternehmens in einem Markt für Massensoftware und soll nun als Grundlage für ein zusätzliches Geschäft mit Auftragsentwicklungen dienen. Man beachte, daß dem Unternehmen den Einstieg in die Auftragsentwicklung annahmegemäß kein Kostenvorteil ermöglicht, sondern die exklusiven Rechte an der Basiskomponente.

Wie bereits diskutiert gehen der eigentlichen Auftragsentwicklung zwingend bilaterale Verhandlungen voraus. Es ist aber nicht sinnvoll, diese Verhandlungen explizit zu modellieren. Vielmehr wird die von Kalai (1977) vorgeschlagene *asymmetrische Nash-Verhandlungslösung* verwendet, da sie mit den Gleichgewichten zahlreicher nicht-kooperativer Verhandlungsspiele korrespondiert. Sie postuliert, daß ein Verhandlungsergebnis sich durch individuelle Rationalität, Pareto-Effizienz, Unabhängigkeit von äquivalenten Nutzentransformationen und Unabhängigkeit von irrelevanten Alternativen auszeichnet.

Da das Verhandlungsergebnis Pareto-effizient ist, maximiert es zwingend den gemeinsamen Nutzen

$$U_A(\theta, e) = u(\theta, b_A, e) - c(e) = \theta \cdot (1 + e) - e^2/2. \tag{4.44}$$

Die Verhandlungen werden also immer zum effizienten Qualitätsniveau $e_A^*(\theta) = \theta$ und dem Gesamtnutzen $U_A^*(\theta) = \theta + \theta^2/2$ führen. Es gilt nun, die gesamte

Verhandlungsmasse $U_A^*(\theta)$ in der Form von Gewinn g und Konsumentenrente k auf die Verhandlungspartner aufzuteilen. Wie diese Aufteilung erfolgt, hängt im Fall der asymmetrischen Nash-Verhandlungslösung von einem weiteren Parameter $m \in [0,1]$ ab, der die *Verhandlungsmacht* des Softwareunternehmens gegenüber den Nutzern beschreibt. Dabei liegt für $m = 0$ alle Macht bei den Nutzern und für $m = 1$ beim Unternehmen; für $m = 1/2$ ist sie hingegen symmetrisch verteilt. Einen weiteren Einfluß hat, daß jeder Nutzer bei einem Scheitern der Verhandlungen auf dem Weg der Eigenentwicklung das Nutzenniveau $U_E^*(\theta)$ erreichen kann. Das eindeutige Verhandlungsergebnis ergibt sich nun nach Kalai (1977) als Lösung des folgenden Maximierungsproblems:

$$\begin{aligned} \max_{g,k} \quad & g^m \cdot (k - U_E^*(\theta))^{1-m} \\ \text{u.d.N.} \quad & g + k = U_A^*(\theta) \end{aligned} \tag{4.45}$$

Es lautet, wie leicht zu zeigen ist, $g^*(\theta) = -m \cdot U_E^*(\theta) + m \cdot U_A^*(\theta) = m \cdot (\theta - \theta^2/2)$ und $k^*(\theta) = m \cdot U_E^*(\theta) + (1-m) \cdot U_A^*(\theta) = \theta + \theta^2/2 - m \cdot (\theta - \theta^2/2)$. Offensichtlich teilen die beiden Verhandlungspartner den durch ihre Einigung erzeugten Mehrwert $U_A^*(\theta) - U_E^*(\theta) = \theta - \theta^2/2$ gemäß ihrer Verhandlungsmacht.

Das Unternehmen bedient alle Nutzer und erzielt den Gewinn

$$G_A^*(m) = \int_0^1 g^*(\theta)\, d\theta = m/3 \tag{4.46}$$

und die Wohlfahrt erreicht den sozial optimalen Wert

$$W_A^* = \int_0^1 U_A^*(\theta)\, d\theta = 2/3. \tag{4.47}$$

Alle Nutzer erhalten nach effizienten Verhandlungen Zugriff auf die qualitativ hochwertige Basiskomponente des Unternehmens sowie eine individuelle Erweiterungskomponente sozial effizienter Qualität.

4.3.3.3 API-unterstützte Eigenentwicklung

Auch wenn Auftragsentwicklung in Abwesenheit asymmetrischer Informationen und unvollständiger Verträge effizient ist, erzielt das Unternehmen nach

Gleichung (4.46) nur einen mageren Gewinn, wenn es über wenig Verhandlungsmacht gegenüber den Nutzern verfügt.[22] Es mag sich daher entschließen, seine exklusive Basiskomponente nicht im Rahmen von Auftragsentwicklungen anzubieten, sondern als Grundlage für API-unterstützte Eigenentwicklungen. Dazu modifiziert das Unternehmen die Basiskomponente so, daß sie jeder Nutzer in Eigenregie an seine spezifischen Bedürfnisse anpassen kann, und verkauft sie dann für Lizenzgebühren in Höhe von $l \geq 0$. Vereinfachend wird dabei jede Möglichkeit zur Preisdiskriminierung ausgeschlossen.

Erwirbt ein Nutzer das API, so wird er seinen Nutzen

$$U_{API}(\theta, e, l) = u(\theta, b_A, e) - c(e) - l = \theta \cdot (1 + e) - e^2/2 - l \tag{4.48}$$

maximieren, indem er das Qualitätsniveau $e^*_{API}(\theta) = \theta$ wählt, und so das Nutzenniveau $U^*_{API}(\theta, l) = \theta + \theta^2/2 - l$ erreichen. Da aber weiterhin die Möglichkeit der Eigenentwicklung besteht, werden nur all jene Nutzer das API tatsächlich kaufen, für die $U^*_{API}(\theta, l) \geq U^*_E(\theta)$ gilt. Es ist leicht zu überprüfen, daß dies alle Nutzer aus dem Intervall $[\bar{\theta}(l), 1]$ mit $\bar{\theta}(l) = 1 - \sqrt{1 - 2l}$ als indifferentem Nutzer sind.[23]

Das Unternehmen maximiert dann seinen Gewinn $G_{API}(l) = l \cdot (1 - \bar{\theta}(l)) = l \cdot \sqrt{1 - 2l}$, indem es Lizenzgebühren in Höhe von $l^* = 1/3$ erhebt. Daraus resultieren auch der gleichgewichtige indifferente Konsument $\theta^* = \bar{\theta}(l^*) = 1 - \sqrt{1/3} \approx 0,42$ und der Gleichgewichtsgewinn $G^*_{API} = G_{API}(l^*) = \sqrt{1/27} \approx 0,19$.

Die Wohlfahrt beträgt

$$W^*_{API} = \int_0^{\theta^*} U^*_E(\theta)\, d\theta + \int_{\theta^*}^1 U^*_{API}(\theta, l^*)\, d\theta + G^*_{API} \approx 0,59. \tag{4.49}$$

[22] Für die asymmetrische Nash-Verhandlungslösung ist die Verhandlungsmacht ein exogener Parameter. Die nicht-kooperative Spieltheorie hat aber mehrere mögliche Bestimmungsgrößen identifizert, etwa die relative Ungeduld und Informiertheit der Verhandlungsparteien.

[23] Der indifferente Nutzer $\bar{\theta}(l)$ ergibt sich nur für $l \in [0, 1/2]$ als einzige Lösung der Gleichung $U^*_{API}(\theta, l) = U^*_E(\theta)$ im Intervall $[0, 1]$. Für $l \notin [0, 1/2]$ existiert hingegen kein indifferenter Konsument und alle Nutzer entscheiden sich für dieselbe Option.

Dabei erfaßt das erste Integral die Konsumentenrente all jener Nutzer, die echte Eigenentwicklung betreiben, und das zweite Integral die Konsumentenrente der Käufer des API. Man beachte, daß $W^*_{API} < W^*_A$ ist, da nicht alle die qualitativ hochwertige Basiskomponente nutzen können.

4.3.3.4 Offene Software

Liegt die Basiskomponente der Qualität $b_A = 1$ hingegen als offene Software vor, so kann sie von allen Nutzern quasi als kostenloses API genutzt werden. Die Wohlfahrt erreicht dann wieder den sozial optimalen Wert

$$W^*_O = \int_0^1 U^*_{API}(\theta, 0)\, d\theta = 2/3. \tag{4.50}$$

4.3.4 Schlußfolgerungen

Das Modell kommt zu zwei wesentlichen Ergebnissen: Erstens ist es immer wünschenswert, wenn eine für einen Massenmarkt entwickelte, qualitativ hochwertige Basiskomponente auch für die Entwicklung von Individualsoftware zur Verfügung steht, und zwar unabhängig davon, wie die Eigentumsrechte an dieser Komponente ausgestaltet sind und in welcher Form sie den Nutzern bereitgestellt wird. Redundante Entwicklungsarbeit und die Verwendung minderwertiger Basiskomponenten werden eingeschränkt. Der folgende Satz faßt diese Aussage zusammen:

Satz 4.6 *Die Verfügbarkeit einer hochwertigen Basiskomponente für die Entwicklung von Individualsoftware wirkt in jedem Fall wohlfahrtssteigernd, da gilt:*

$$W^*_E < \min\{W^*_{API}, W^*_A, W^*_O\}.$$

Zweitens zeigt sich, daß ein Unternehmen eine von ihm kontrollierte Basiskomponente unter gewissen Bedingungen ineffizent verbreitet, selbst wenn effiziente Verträge möglich sind. Denn es gilt:

Satz 4.7 *Liegen die Eigentumsrechte für die Basiskomponente bei einem Unternehmen, so wird dieses bei einer hinreichend geringen Verhandlungsmacht m die Basiskomponente wegen $G^*_{API} > G^*_A(m)$ nicht effizient auf dem Weg der Auftragsentwicklung anbieten, sondern ineffizient als API. Im Fall offener Software findet die Basiskomponente hingegen immer eine effiziente Verbreitung.*

Die schiere Existenz offener Software hat also eine gesamtgesellschaftlich wünschenswerte Wirkung auf Märkte für Individualsoftware. Dieser Umstand sollte bei staatlichen Maßnahmen mit einem Bezug zu offener Software angemessen berücksichtigt werden.

4.4 Zwischenfazit

Im Wettbewerb zwischen proprietärer und offener Massensoftware konnte kein Marktversagen festgestellt werden, welches ein direktes Eingreifen des Staates in das Marktgeschehen zugunsten offener Software durch Subventionen oder Zwang rechtfertigt. Allenfalls dann, wenn die proprietäre Softwareindustrie im Ausland angesiedelt ist, mögen derartige Instrumente zur Verlagerung von Renten ins Inland dienen. Für ein Land wie Deutschland, welches sich politisch dem Freihandel verpflichtet fühlt, ist ein derartiges Vorgehen aber sicherlich undenkbar. Dennoch ist das Aufkommen offener Software gesamtgesellschaftlich erfreulich, da sie den Wettbewerb bereichert und mögliche Ineffizienzen bei der quantitativ bedeutsamen Entwicklung von Individualsoftware zu mildern hilft. Insgesamt sollte der Staat sich daher darauf beschränken, einen geeigneten Rahmen für den ungehinderten Wettbewerb zwischen proprietärer und offener Software abzustecken. Direkte Eingriffe sind hingegen abzulehnen.

5 Schlußbetrachtung

In diesem Kapitel werden zunächst die Hauptergebnisse der vorliegenden Arbeit zusammengefaßt. Anschließend werden einige interessante Fragestellungen für weiterführende Forschungen vorgestellt.

5.1 Zusammenfassung der Hauptergebnisse

Insgesamt erweist sich offene Software als ein äußerst vielfältiges Phänomen, bei dem insbesondere die einzigartige Nutzung geistiger Eigentumsrechte und der offene Entwicklungsprozeß hervorstechen. Bemerkenswert ist ihre Entwicklung deshalb, weil diese trotz der fundamentalen Andersartigkeit gegenüber proprietärer Softwareentwicklung regelmäßig vollwertige, konkurrenzfähige Produkte hervorbringt.

Anders, als es zunächst den Anschein hat, gründet sich der Erfolg offener Softwareentwicklung dabei auf eine Vielzahl handfester ökonomischer Motive. Die entscheidende Frage ist dann, unter welchen Voraussetzungen es den teilweise sehr viele Mitglieder umfassenden Entwicklergemeinschaften gelingt, ihr Handeln zu koordinieren und eine offene Software auch langfristig zuverlässig bereitzustellen. Mittels eines geeigneten Modells wurden in Kapitel 3 mehrere hinreichende Bedingungen für die Nachhaltigkeit eines offenen Softwareprojekts herausgearbeitet: Besteht die Software aus vielen, wohldefinierten Modulen, hat sie technisch versierte Nutzer, für welche die Verfügbarkeit der Quelltexte entsprechend wertvoll ist, und zahlt sich erfolgreiche Koordination auch für die einzelnen Entwickler aus, dann kann die Existenz eines offenen Entwicklungsprojekts auch langfristig als gesichert gelten.

Entsprechende Debatten in der politischen Arena werfen zudem die Frage auf, ob und, wenn ja, wie die öffentliche Hand zugunsten offener Software in das Marktgeschehen eingreifen sollte. Im Fall von Massensoftware zeigt sich, daß keine Eigenschaft entsprechender Märkte zu einem korrekturbedürftigen Marktversagen im Wettbewerb zwischen offener und proprietärer Software führt. Im quantitativ bedeutsameren Fall von Individualsoftware sind sinnvolle staatliche Maßnahmen zudem ganz grundsätzlich kaum vorstellbar, da hier entweder in Eigenregie und somit abseits von Märkten entwickelt wird, oder aber auf der Grundlage frei gestalteter Verträge als Auftragsarbeit. Dennoch ist das Aufkommen offener Software an sich zu begrüßen, da sie den Wettbewerb intensiviert und die effiziente Entwicklung von Individualsoftware begünstigt. Insgesamt scheint es geraten, auf eine aktive Förderung offener Software zu verzichten, aber durch eine geeignete Gestaltung der Rahmenbedingungen einen freien Wettbewerb sicherzustellen. Beide Arten von Software haben spezifische Stärken und Schwächen, so daß sie auf lange Sicht koexistieren werden. Die von einigen Beobachtern prophezeite Revolution der Softwareindustrie fällt aus.

5.2 Ausblick

Der vielversprechendste Gegenstand weiterführender Untersuchungen ist die Mikrostruktur offener Softwareprojekte, die zwischen den verschiedenen Projekten stärker variiert, als dies von allen bislang vorgelegten Modellen erfaßt wird. Wie die Wahl der offenen Lizenz, die Organisationsstruktur und die Ausgestaltung von Führung und Entscheidungsgewalt ineinandergreifen und auf verschiedene Arten von Entwicklern und Konsumenten wirken, ist aber in vielerlei Hinsicht relevant. Wie etwa soll der Gründer eines Entwicklungsprojekts die Mikrostruktur gestalten, damit das Projekt seinen Zielen dient? Wie steht es um die langfristige Verträglichkeit kommerzieller und nicht-kommerzieller Interessen? Und welche Wechselwirkungen existieren zwischen dem Erfolg einer offenen Software in Massenmärkten und ihrer zukünftigen Entwicklung? Ein besseres Verständnis gerade der letztgenannten Frage würde es ermöglichen,

die in Kapitel 3 vorgestellten Modelle zur Entwicklung offener Software und die in Kapitel 4 behandelten zu ihrer Nutzung besser als bisher zu verknüpfen und so ein umfassenderes Bild der ökonomischen Aspekte offener Software zu erlangen.

Des weiteren ist offene Software im Hinblick auf neue Unternehmensstrategien zu untersuchen. So kann sie als eine besonders glaubhafte Form der Selbstverpflichtung gegenüber Konsumenten, einen Lock-In nicht auszunutzen, als Gestaltungsmittel einer Kooperation mehrerer Unternehmen und als Ansatz zur Einführung offener Standards eingesetzt werden. Ob und wann diese Verwendungen offener Software traditionellen strategischen Mitteln wie etwa Entwicklungskonsortien, Patentpools und Standardisierungsorganisationen vorzuziehen sind, ist derzeit nur unbefriedigend geklärt.

Grundsätzlicherer Natur ist die Frage, ob die Prinzipien offener Softwareentwicklung auf andere Industriezweige übertragbar sind. So wird derzeit diskutiert, auch Hardware in einem offenen Prozeß zu entwickeln, indem etwa die Schaltkreis-Layouts von Mikroprozessoren unter geeignet modifizierten offenen Lizenzen veröffentlicht werden. Ein solches Vorgehen wäre grundsätzlich möglich, da auch die exakte Spezifikation von Hardware sehr einfach eindeutig und verlustfrei zugänglich gemacht und auch verändert werden kann. Als unüberwindbares Hindernis mag sich aber erweisen, daß die Umsetzung der logischen Produktspezifikation in ein nutzbares Produkt bei Hardware kostspieliger Fertigungsanlagen bedarf, während im Fall von Software lediglich preiswerte oder gar kostenlos verfügbare Entwicklungswerkzeuge erforderlich sind. Ähnliche Überlegungen und potentielle Schwierigkeiten findet man auch im Bereich der Biotechnologie und Genforschung. Es ist daher interessant zu klären, welche Charakteristika von Software tatsächlich notwendige Voraussetzungen für eine erfolgreiche Produktentwicklung à la offener Software sind.

Des weiteren verspricht die eingehendere Untersuchung offener Softwareentwicklung Einsichten dazu, wie die vielbeschworenen Wissensgesellschaften der Zukunft ihre gewerblichen Schutz- und Urheberrechte ausgestalten sollten. Das eine Extrem bilden starke Rechte, welche einem Urheber die vollkommene Kon-

trolle über die Verwendung seines geistigen Eigentums ermöglichen, aber auch zukünftige Innovationen behindern, das andere hingegen die vollkommene Rechtlosigkeit und der damit einhergehende Wegfall fast aller Innovationsanreize. Inwieweit offene Software, die ja einen neuartigen Mittelweg zwischen diesen beiden Extremen aufzeigt, diesbezüglich ein Vorbild darstellt, gilt es zu ergründen.

A Die Open Source Definition (OSD)

Nachfolgend ist die Open Source Definition (OSD) in der Version 1.9 angegeben. Diese, derzeit aktuelle Version sowie eine Liste konkreter Lizenzen, welche der OSD genügen, finden sich unter `www.opensource.org`.

The Open Source Definition Introduction Open source doesn't just mean access to the source code. The distribution terms of open-source software must comply with the following criteria:

1. Free Redistribution The license shall not restrict any party from selling or giving away the software as a component of an aggregate software distribution containing programs from several different sources. The license shall not require a royalty or other fee for such sale.

2. Source Code The program must include source code, and must allow distribution in source code as well as compiled form. Where some form of a product is not distributed with source code, there must be a well-publicized means of obtaining the source code for no more than a reasonable reproduction cost – preferably, downloading via the Internet without charge. The source code must be the preferred form in which a programmer would modify the program. Deliberately obfuscated source code is not allowed. Intermediate forms such as the output of a preprocessor or translator are not allowed.

3. Derived Works The license must allow modifications and derived works, and must allow them to be distributed under the same terms as the license of the original software.

4. **Integrity of The Author's Source Code** The license may restrict source-code from being distributed in modified form *only* if the license allows the distribution of "patch files" with the source code for the purpose of modifying the program at build time. The license must explicitly permit distribution of software built from modified source code. The license may require derived works to carry a different name or version number from the original software.

5. **No Discrimination Against Persons or Groups** The license must not discriminate against any person or group of persons.

6. **No Discrimination Against Fields of Endeavor** The license must not restrict anyone from making use of the program in a specific field of endeavor. For example, it may not restrict the program from being used in a business, or from being used for genetic research.

7. **Distribution of License** The rights attached to the program must apply to all to whom the program is redistributed without the need for execution of an additional license by those parties.

8. **License Must Not Be Specific to a Product** The rights attached to the program must not depend on the program's being part of a particular software distribution. If the program is extracted from that distribution and used or distributed within the terms of the program's license, all parties to whom the program is redistributed should have the same rights as those that are granted in conjunction with the original software distribution.

9. **License Must Not Restrict Other Software** The license must not place restrictions on other software that is distributed along with the licensed software. For example, the license must not insist that all other programs distributed on the same medium must be open-source software.

10. **License Must Be Technology-Neutral** No provision of the license may be predicated on any individual technology or style of interface.

B Angebotsstruktur offener Software

Nachfolgend wird die Datengrundlage für die in Tabelle 2.9 beschriebene Struktur des Angebots an offener Software angegeben:

Quelle	Originalkategorie	Anzahl	Kategorie
FreshMeat	Adaptive Technologies	42	Sonstiges
FreshMeat	Artistic Software	142	Anwendungen
FreshMeat	Communications	5436	Netzwerkmanagement
FreshMeat	Database	1902	Informationsmanagement
FreshMeat	Desktop Environment	2225	System
FreshMeat	Documentation	155	Sonstiges
FreshMeat	Education	830	Anwendungen
FreshMeat	freshmeat.net	28	Sonstiges
FreshMeat	Games/Entertainment	2548	Anwendungen
FreshMeat	Home Automation	71	Anwendungen
FreshMeat	Information Management	1073	Informationsmanagement
FreshMeat	Internet	9653	Netzwerkmanagement
FreshMeat	Multimedia	5867	Anwendungen
FreshMeat	Office/Business	1896	Anwendungen
FreshMeat	Other/Nonlisted Topic	252	Sonstiges
FreshMeat	Printing	221	System
FreshMeat	Religion	47	Sonstiges
FreshMeat	Scientific/Engineering	2530	Anwendungen
FreshMeat	Security	1510	System
FreshMeat	Software Development	9990	Programmentwicklung
FreshMeat	System	9714	System
FreshMeat	Terminals	304	Anwendungen
FreshMeat	Text Editors	615	Anwendungen
FreshMeat	Text Processing	2231	Anwendungen
FreshMeat	Utilities	2926	Anwendungen
SourceForge	Communications	11155	Netzwerkmanagement
SourceForge	Database	4640	Informationsmanagement
SourceForge	Desktop Environment	2422	System
SourceForge	Education	2400	Anwendungen
SourceForge	Formats and Protocols	144	Sonstiges
SourceForge	Games/Entertainment	10456	Anwendungen
SourceForge	Internet	18206	Netzwerkmanagement
SourceForge	Multimedia	9315	Anwendungen
SourceForge	Office/Business	3875	Anwendungen
SourceForge	Other/Nonlisted Topic	1825	Sonstiges
SourceForge	Printing	350	System
SourceForge	Religion and Philosophy	216	Anwendungen
SourceForge	Scientific/Engineering	7387	Anwendungen
SourceForge	Security	2004	System
SourceForge	Sociology	293	Anwendungen
SourceForge	Software Development	13065	Programmentwicklung
SourceForge	System	14700	System

Quelle	Originalkategorie	Anzahl	Kategorie
SourceForge	Terminals	464	Anwendungen
SourceForge	Text Editors	2054	Anwendungen
RedHat	System Environment	177	System
RedHat	Applications	162	Anwendungen
RedHat	Development	120	Programmentwicklung
RedHat	User Interface	56	System
RedHat	Documentation	35	Sonstiges
RedHat	Amusements	28	Anwendungen
SourceWell	Console-Backup	43	System
SourceWell	Console-CD or DVD Writing	11	Anwendungen
SourceWell	Console-Databases	23	Informationsmanagement
SourceWell	Console-Document Processing	20	Anwendungen
SourceWell	Console-Editors	21	Anwendungen
SourceWell	Console-Emulators	8	System
SourceWell	Console-File & Disk Management	35	System
SourceWell	Console-Games	21	Anwendungen
SourceWell	Console-Graphics	7	Anwendungen
SourceWell	Console-Image Processing	2	Anwendungen
SourceWell	Console-Kernel Drivers & Modules	24	System
SourceWell	Console-Miscellaneous	14	Sonstiges
SourceWell	Console-Multimedia	14	Anwendungen
SourceWell	Console-Operating Systems	107	System
SourceWell	Console-Personal Tools	8	Anwendungen
SourceWell	Console-Printing	3	System
SourceWell	Console-Science & Mathematic	9	Anwendungen
SourceWell	Console-Security	12	System
SourceWell	Console-Shells	6	System
SourceWell	Console-Sound	19	System
SourceWell	Console-Utilities	84	System
SourceWell	Console-Video	10	System
SourceWell	Development-Application & Software Development	38	Programmentwicklung
SourceWell	Development-Compiler	11	Programmentwicklung
SourceWell	Development-Database	8	Programmentwicklung
SourceWell	Development-Debugging	10	Programmentwicklung
SourceWell	Development-Documentation	2	Sonstiges
SourceWell	Development-IDE	10	Programmentwicklung
SourceWell	Development-Interpreter	10	Programmentwicklung
SourceWell	Development-Languages	23	Programmentwicklung
SourceWell	Development-Libraries & Classes	78	Programmentwicklung
SourceWell	Development-Miscellaneous	16	Sonstiges
SourceWell	Development-Revision Control	26	Programmentwicklung
SourceWell	Development-Translator	0	Programmentwicklung
SourceWell	Development-Utilities	30	Programmentwicklung
SourceWell	GNOME-Applications	89	Anwendungen
SourceWell	GNOME-CD or DVD Writing	0	Anwendungen
SourceWell	GNOME-Core	19	System
SourceWell	GNOME-Databases	5	Informationsmanagement
SourceWell	GNOME-Desktop	6	System
SourceWell	GNOME-Development	89	Programmentwicklung
SourceWell	GNOME-Games	19	Anwendungen
SourceWell	GNOME-Graphics	14	Anwendungen
SourceWell	GNOME-Miscellaneous	20	Sonstiges
SourceWell	GNOME-Multimedia	15	Anwendungen
SourceWell	GNOME-Networking	66	Netzwerkmanagement
SourceWell	GNOME-Office	2	Anwendungen
SourceWell	GNOME-Sound	23	Anwendungen
SourceWell	GNOME-System Utilities	27	System
SourceWell	GNOME-Utilities	16	Anwendungen
SourceWell	GNOME-Video	6	Anwendungen
SourceWell	KDE-Applications	87	Anwendungen
SourceWell	KDE-CD or DVD Writing	9	Anwendungen

Quelle	Originalkategorie	Anzahl	Kategorie
SourceWell	KDE-Chat	3	Anwendungen
SourceWell	KDE-Core	23	System
SourceWell	KDE-Databases	8	Informationsmanagement
SourceWell	KDE-Desktop	10	System
SourceWell	KDE-Development	54	Programmentwicklung
SourceWell	KDE-Education	14	Anwendungen
SourceWell	KDE-Games	37	Anwendungen
SourceWell	KDE-Graphics	22	Anwendungen
SourceWell	KDE-Miscellaneous	12	Sonstiges
SourceWell	KDE-Multimedia	15	Anwendungen
SourceWell	KDE-Networking	99	Netzwerkmanagement
SourceWell	KDE-Office	24	Anwendungen
SourceWell	KDE-Sound	33	Anwendungen
SourceWell	KDE-System Utilities	30	System
SourceWell	KDE-Utilities	47	Anwendungen
SourceWell	KDE-Video	7	Anwendungen
SourceWell	Networking-Administration	37	Netzwerkmanagement
SourceWell	Networking-CGI	9	Netzwerkmanagement
SourceWell	Networking-Chat	27	Netzwerkmanagement
SourceWell	Networking-Clients	6	Anwendungen
SourceWell	Networking-Cluster	7	Netzwerkmanagement
SourceWell	Networking-Conferencing	6	Netzwerkmanagement
SourceWell	Networking-Content Management System	35	Netzwerkmanagement
SourceWell	Networking-Cooperative Work	20	Netzwerkmanagement
SourceWell	Networking-Customer Relationship Management	1	Netzwerkmanagement
SourceWell	Networking-Directory	7	Netzwerkmanagement
SourceWell	Networking-E-Commerce & E-Business	15	Netzwerkmanagement
SourceWell	Networking-E-Mail	55	Netzwerkmanagement
SourceWell	Networking-Fax	2	Netzwerkmanagement
SourceWell	Networking-File Sharing	4	Netzwerkmanagement
SourceWell	Networking-File Transfer	29	Netzwerkmanagement
SourceWell	Networking-Financial	3	Netzwerkmanagement
SourceWell	Networking-HTML Tool	22	Netzwerkmanagement
SourceWell	Networking-IP Telephony	1	Netzwerkmanagement
SourceWell	Networking-Log Analyzers	9	Netzwerkmanagement
SourceWell	Networking-Middleware	11	Netzwerkmanagement
SourceWell	Networking-Mirroring	13	Netzwerkmanagement
SourceWell	Networking-Miscellaneous	28	Sonstiges
SourceWell	Networking-Monitoring	29	Netzwerkmanagement
SourceWell	Networking-News	8	Netzwerkmanagement
SourceWell	Networking-Peer-to-Peer	18	Netzwerkmanagement
SourceWell	Networking-PHP	112	Netzwerkmanagement
SourceWell	Networking-Printing	2	Netzwerkmanagement
SourceWell	Networking-Proxy Server	7	Netzwerkmanagement
SourceWell	Networking-Remote Control	7	Netzwerkmanagement
SourceWell	Networking-Remote Login	4	Netzwerkmanagement
SourceWell	Networking-Router,Firewall and Security	15	Netzwerkmanagement
SourceWell	Networking-Search Engine	17	Netzwerkmanagement
SourceWell	Networking-Servers	26	Netzwerkmanagement
SourceWell	Networking-SMS & WAP	10	Netzwerkmanagement
SourceWell	Networking-Sound	6	Netzwerkmanagement
SourceWell	Networking-Unified Messaging	2	Netzwerkmanagement
SourceWell	Networking-Video	5	Netzwerkmanagement
SourceWell	Networking-Web Application	34	Anwendungen
SourceWell	Networking-Web Browser	24	Anwendungen
SourceWell	Networking-Web Browser Plug-ins	8	Anwendungen
SourceWell	Networking-Web Development	38	Programmentwicklung
SourceWell	Networking-Web Server	21	Netzwerkmanagement
SourceWell	Networking-XML Tool	9	Netzwerkmanagement
SourceWell	X11-Backup	0	System
SourceWell	X11-CD or DVD Writing	0	Anwendungen

Quelle	Originalkategorie	Anzahl	Kategorie
SourceWell	X11-Databases	9	Informationsmanagement
SourceWell	X11-Desktop, File & Disk Managers	8	Anwendungen
SourceWell	X11-Document Processing	7	Anwendungen
SourceWell	X11-Editors	10	Anwendungen
SourceWell	X11-Emulators	4	System
SourceWell	X11-Financial	6	Anwendungen
SourceWell	X11-Games	20	Anwendungen
SourceWell	X11-Graphics	26	Anwendungen
SourceWell	X11-GUI Builder	1	Programmentwicklung
SourceWell	X11-Image Processing	6	Anwendungen
SourceWell	X11-Java	7	Programmentwicklung
SourceWell	X11-Miscellaneous	18	Sonstiges
SourceWell	X11-Multimedia	16	Anwendungen
SourceWell	X11-Office	9	Anwendungen
SourceWell	X11-Personal Desktop Tools	35	Anwendungen
SourceWell	X11-Science & Mathematic	11	Anwendungen
SourceWell	X11-Server	1	System
SourceWell	X11-Sound	18	Anwendungen
SourceWell	X11-System	7	System
SourceWell	X11-System Utilities	14	System
SourceWell	X11-Video	11	Anwendungen
SourceWell	X11-Window Manager	21	System
SoftwareHouse	Backup	25	System
SoftwareHouse	Betriebssysteme	13	System
SoftwareHouse	Bibliographie	5	Anwendungen
SoftwareHouse	Bücher	65	Sonstiges
SoftwareHouse	Buchhaltung	15	Anwendungen
SoftwareHouse	Büroapplikationen	131	Anwendungen
SoftwareHouse	CAD	7	Anwendungen
SoftwareHouse	CD Brenner	12	Anwendungen
SoftwareHouse	Chemie	6	Anwendungen
SoftwareHouse	Components	4	Programmentwicklung
SoftwareHouse	Datenanalyse und Visualisierung	14	Anwendungen
SoftwareHouse	Datenbanken	24	Informationsmanagement
SoftwareHouse	E-Mail/Groupware	25	Anwendungen
SoftwareHouse	eBusiness	26	Informationsmanagement
SoftwareHouse	Edutainment	4	Anwendungen
SoftwareHouse	Finanzen	14	Anwendungen
SoftwareHouse	Grafik	140	Anwendungen
SoftwareHouse	Grafik Plugins	13	Anwendungen
SoftwareHouse	Heim & Hobby	48	Anwendungen
SoftwareHouse	Internet	123	Netzwerkmanagement
SoftwareHouse	Kommunikation	39	Anwendungen
SoftwareHouse	Mathematik	22	Anwendungen
SoftwareHouse	Multimedia	27	Anwendungen
SoftwareHouse	Musik	37	Anwendungen
SoftwareHouse	Nachschlagewerke	25	Anwendungen
SoftwareHouse	Netzwerk	69	Netzwerkmanagement
SoftwareHouse	Projektmanagement	4	Anwendungen
SoftwareHouse	Reisen	15	Anwendungen
SoftwareHouse	Software für Kinder	1	Anwendungen
SoftwareHouse	Software Management	6	System
SoftwareHouse	Softwareentwicklung	62	Programmentwicklung
SoftwareHouse	Sprachen	26	Anwendungen
SoftwareHouse	Statistik	8	Anwendungen
SoftwareHouse	Texterkennung (OCR)	15	Anwendungen
SoftwareHouse	Tools	227	Anwendungen
SoftwareHouse	Unterhaltung	2	Anwendungen
SoftwareHouse	Verschiedenes	10	Sonstiges
SoftwareHouse	Virenscanner	35	System
SoftwareHouse	Web Design	51	Anwendungen
SoftwareHouse	Wissenschaft	18	Anwendungen

Literaturverzeichnis

Arrow, K. (1962), Economic Welfare and the Allocation of Resources for Invention. In R. Nelson (Hrsg.), *The Rate and Direction of Inventive Activity*, Princeton, Princeton University Press,: 609–625.

Ausubel, L., P. Cramton und R. J. Deneckere (2002), Bargaining with Incomplete Information. In R. Aumann und S. Hart (Hrsg.), *Handbook of Game Theory*, Band 3, Amsterdam, Elsevier Science, 1897–1945.

Backhaus, K., B. Erichson, W. Plinke und R. Weiber (2005), *Multivariate Analysemethoden*, 10. Auflage, Berlin, Springer.

Behlendorf, B. (1999), Open Source as a Business Strategy. In C. DiBona, S. Ockman und M. Stone (Hrsg.), *Open Sources: Voices from the Open Source Revolution*, Sebastopol, O'Reilly, 149–170.

Bergin, J. und B. Lipman (1996), Evolution with State-Dependent Mutations, *Econometrica* **64**, 943–956.

Bergstrom, T. und O. Stark (1993), How Altruism can Prevail in an Evolutionary Environment, *American Economic Review* **83**, 149–155.

Bergquist, M. und J. Ljungberg (2001), The Power of Gifts: Organizing Social Relationships in Open Source Communities, *Information Systems Journal* **11**, 305–320.

Bessen, J. (2005), Open Source Software: Free Provision of Complex Public Goods, Mimeo, School of Law, Boston University.

Bessen, J. und R. Hunt (2004), An Empirical Look at Software Patents, Working Paper No. 03-17/R, Federal Reserve Bank of Philadelphia.

Bessen, J. und E. Maskin (2000), Sequential Innovation, Patents, and Imitation, Working Paper 00-01, Department of Economics, Massachusetts Institute of Technology, Cambridge.

Bezroukov, N. (1999), Open Source Software Development as a Special Type of Academic Research (Critique of Vulgar Raymondism), *First Monday* **4**.

Bitzer, J. und P. Schröder (2005), Bug-Fixing and Code-Writing: The Private Provision of Open Source Software, *Information Economics and Policy* **17**, 389–406.

Boehm, B. (1988), A Spiral Model of Software Development and Enhancement, *IEEE Computer* **21**, 61–72.

Bonaccorsi, A. und C. Rossi (2003), Why Open Source Software can Succeed, *Research Policy* **32**, 1243–1258.

Bonaccorsi, A. und C. Rossi (2005), Intrinsic Motivations and Profit-Oriented Firms in Open Source Software: Do Firms Practise what they Preach?, Mimeo, Sant'Anna School of Advanced Studies, Pisa.

Bozman, J., M. Eastwood, A. Gillen, S. Josselyn, D. Kusnetzky, L. Loverde, V. Turner (2004), *Worldwide Linux 2004-2008 Forecast: Moving from Niche to Mainstream*, Marktanalyse, Dokumentnr. 32424, IDC Software Consulting, Framingham.

Broersma, M. (2004), Open Source Market Share to Double, *techworld.com*, 9. Juli 2004.

Brügge, B., D. Harhoff, A. Picot, O. Creighton, M. Fiedler, J. Henkel (2004), *Open-Source-Software: Eine ökonomische und technische Analyse*, Berlin, Springer.

Brynjolfsson, E. und C. Kemerer (1996), Network Externalities in Microcomputer Software: An Econometric Analysis of the Spreadsheet Marktet, *Management Science* **42**, 1627–1647.

Bundesministerium des Innern (2001), *Empfehlung zur Durchführung von Wirtschaftlichkeitsbetrachtungen in der Bundesverwaltung, insbesondere beim Einsatz der IT*, Schriftenreihe der KBSt, Band 52.

Bundesministerium des Innern (2002), Bundesinnenministerium stellt neue Anwendungen vor: Sichere E-Mails mit Open Source Software versenden, Pressemitteilung vom 17. März 2002.

Bureau of Economic Analysis (2005), Prices and Output for Information and Communication Technologies, U. S. Department of Commerce, Washington. (www.bea.gov/bea/dn/info_comm_tech.htm)

Caillaud, B. und B. Jullien (2001), Software and the Internet: Competing Cybermediaries, *European Economic Review* **45**, 797–808.

Caillaud, B. und B. Jullien (2003), Chicken & Egg: Competing Matchmakers, *RAND Journal of Economics* **34**, 309–328.

Center for Strategic and International Studies (2004), Global Policies on Open Source Software, Studie, Washington. (www.csis.org/tech/OpenSource)

CDU (2002), Chancen@Deutschland: Eine Internetstrategie für die Politik, Beschluss des Bundesvorstands der CDU Deutschlands am 3. Juni 2002. (www.cdu.de/presse/archiv-2002/chancen.pdf)

Comino, S. und F. Manenti (2005), Government Policies Supporting Open Source Software for the Mass Market, *Review of Industrial Organization* **26**, 217–240.

Constant, D., L. Sproull und S. Kiesler (1996), The Kindness of Strangers: On the Usefulness of Electronic Weak Ties for Technical Advice, *Organization Science* **7**, 119–135.

Cusumano, M. (1991), *Japan's Software Factories: A Challenge to U. S. Management*, New York, Oxford University Press.

Cusumano, M. und R. Selby (1991), *Microsoft Secrets: How the World's Most Powerful Software Company Creates Technology, Shapes Markets, and Manages People*, New York, Simon and Schuster.

d'Aspremont, C., J. Gabszewicz und J.-F. Thisse (1979), On Hotelling's Stability in Competition, *Econometrica* **14**, 1145–1151.

Dahlander, L. (2004), Appropriating Returns From Open Innovation Processes: A Multiple Case Study of Small Firms in Open Source Software, Mimeo, Department of Industrial Dynamics, School of Technology Management and Economics, Chalmers University of Technology.

D'Antoni, H. (2004), Open-Source Software Use Joins The Mix, *informationweek.com*, 1. November 2004.

Deutscher Bundestag (2001), Plenarprotokoll 14/152 der 152. Sitzung des Deutschen Bundestags, Berlin. (`dip.bundestag.de/btp/14/14152.pdf`)

Duffy, J. und N. Feltovich (1999), Does Observation of Others Affect Learning in Strategic Environments? An Experimental Study, *International Journal of Game Theory* **28**, 131–152.

Economides, N. (1989), Desirability of Compatibility in the Absence of Network Externalities, *American Economic Review* **79**, 1165–1181.

Economides, N. (1996), The Economics of Networks, *International Journal of Industrial Organization* **16**, 673–699.

Edwards, K. (2001), Epistemic Communities, Situated Learning and Open Source Software Development, Mimeo, Technical University of Denmark, Lyngby.

Edwards, K. (2003), Technological Innovation in Software Industry: Open Source Software Development, Dissertation, Technical University of Denmark, Lyngby.

Ellison, G. (1993), Learning, Local Interaction, and Coordination, *Econometrica* **61**, 1047–1072.

Eshel, I., L. Samuelson und A. Shaked (1998), Altruists, Egoists, and Hooligans in a Local Interaction Model, *American Economic Review* **88**, 157–179.

Ettrich, M. (2004), Koordination und Kommunikation in Open-Source- Projekten. In R. Gehring und B. Lutterbeck (Hrsg.), *Open Source Jahrbuch 2004: Zwischen Softwareentwicklung und Gesellschaftsmodell*, Berlin, Lehmanns Media, 179–192.

Evans, D. (2002), Politics and Programming: Government Preferences for Promoting Open Source Software. In R. Hahn (Hrsg.), *Government Policy toward Open Source Software*, Washington, Brookings Institution Press, 34–49.

Farrell, J. und G. Saloner (1985), Standardization, Compatibility, and Innovation, *RAND Journal of Economics* **16**, 70–83.

FDP (2002), OTTO: Linux – soweit es vernünftig ist, Pressemitteilung vom 28. Februar 2002.

Feller, J. und B. Fitzgerald (2002), *Understanding Open Source Software Development*, London, Pearson Education.

Fink, M. (2003), *The Business and Economics of Linux and Open Source*, New Jersey, Pearson Education.

Fosfuri, A. und M. Giarratana (2004), Product Strategies and Startups' Survival in Turbulent Industries: Evidence from the Security Software Industry, Mimeo, Department of Business Administration, Universidad Carlos III de Madrid.

Fudenberg, D. und J. Tirole (1991), *Game Theory*, Cambridge, MIT Press.

Gandal, N. (1994), Hedonic Price Indexes for Spreadsheets and Empirical Test for Network Externalities, *RAND Journal of Economics* **25**, 160–170.

Gandal, N. (2002), Compatibility, Standardization and Network Effects: Some Policy Implications, *Oxford Review of Economic Policy* **18**, 80–91.

Gonsalves, A. (2003), Windows Gains Market Share, Despite Linux Threat, *techweb.com*, 8. Oktober 2003.

Goode, S. (2005), Something for Nothing: Management Rejection of Open Source Software in Australia's Top Firms, *Information and Management* **42**, 669–681.

Gosh, R. und V. Prakash (2000), The Orbiten Free Software Survey, *First Monday* **5**.

Gosh, R., R. Glott, B. Krieger und G. Robles (2002), Free/Libre and Open Source Software, Survey and Study, Part 4: Survey of Developers, International Institute of Infonomics, University of Maastricht.

Gosh, R., G. Robles und R. Glott (2002), Free/Libre and Open Source Software, Survey and Study, Part 5: Software Source Code Survey, International Institute of Infonomics, University of Maastricht.

Gröhn, A. (1999), *Netzwerkeffekte und Wettbewerbspolitik: Eine ökonomische Analyse des Softwaremarktes*, Kieler Studien 296, Tübingen, Mohr Siebeck.

Gutsche, J. (2004), Autonome Produktentwicklung durch Konsumenten: Lehren aus der Open-Source-Bewegung, Mimeo, Fakultät für Volkswirtschaftslehre, Universität Mannheim.

Gutsche, J. (2005a), Competition between Open Source and Proprietary Software, and the Scope for Public Policy. In M. Scotto und G. Succi (Hrsg.), *Proceedings of the 1st International Conference on Open Source Systems*, Genua, ECIG, 196–199.

Gutsche, J. (2005b), The Evolution of Open Source Communities. *Topics in Economic Analysis and Policy* **5**, Artikel 2.

Hahn, R. (2002), Government Policy toward Open Source Software: An Overview. In R. Hahn (Hrsg.), *Government Policy toward Open Source Software*, Washington, Brookings Institution Press, 1–11.

Harhoff, D., J. Henkel und E. von Hippel (2003), Profiting from Voluntary Information Spillovers: How Users Benefit by Freely Revealing their Innovations. *Research Policy* **32**, 1753–1769.

Hars, A. und S. Ou (2002), Working for Free? Motivations for Participating in Open-Source Projects, *International Journal of Electronic Commerce* **6**, 25–39.

Hawkins, R. (2004), The Economics of Open Source Software for a Competitive Firm, *Netnomics* **6**, 103–117.

Henkel, J. (2004a), Open Source Software from Commercial Firms: Tools, Complements, and Collective Invention, *Zeitschrift für Betriebswirtschaft*, Ergänzungsheft 4/2004.

Henkel, J. (2004b), The Jukebox Mode of Innovation: A Model of Commercial Open Source Development, CEPR Discussion Paper 4507, Centre for Economic Policy Research, London.

Henkel, J. (2004c), Patterns of Free Revealing: Balancing Code Sharing and Protection in Commercial Open Source Development, Mimeo, Institute for Innovation Research, Technology Management and Entrepreneurship, Universität München.

Henkel, J. und M. Tins (2004), Munich/MIT Survey: Development of Embedded Linux, Mimeo, Institute for Innovation Research, Technology Management and Entrepreneurship, Universität München.

Hertel, G., S. Niedner und S. Herrmann (2003), Motivation of Software Developers in Open Source Projects: An Internet-based Survey of Contributors to the Linux Kernel, *Research Policy* **32**, 1159–1177.

Heise-Newsticker (2003a), Brasiliens Präsident befürwortet Open Source, *heise.de*, 8. September 2003.

Heise-Newsticker (2003b), China, Südkorea und Japan planen Windows-Ersatz, *heise.de*, 1. September 2003.

Heise-Newsticker (2004), Malaysia setzt verstärkt auf Open Source, *heise.de*, 23. Juli 2004.

Hissam, S. und C. Weinstock (2001), Open Source Software: the Other Commercial Software. In J. Feller, B. Fitzgerald und A. van der Hoek (Hrsg.), *Making Sense of the Bazaar: Proceedings of the 1st Workshop on Open Source Software Engineering*. (opensource.ucc.ie/icse2001)

Hotelling, H. (1929), Stability and Competition, *Economic Journal* **39**, 41–57.

Huck, S., H.-T. Normann und J. Oechssler (1999), Learning in Cournot Oligopoly: An Experiment, *Economic Journal* **109**, 80–95.

Jacobson, I., G. Booch und J. Rumbaugh (1999), *The Unified Software Development Process*. Reading, Addison-Wesley.

Jaeger, T. und A. Metzger (2002), *Open Source Software: Rechtliche Rahmenbedingungen der Freien Software*, München, Beck.

Jeffries, R., A. Anderson und C. Hendrickson (2000), *Extreme Programming Installed*, Reading, Addison-Wesley.

Johnson, J. (2002), Open source software: Private Provision of a Public Good, *Journal of Economics and Management Strategy* **11**, 637–662.

Jørgensen, N. (2001a), Putting it all in the Trunk: Incremental Software Development in the FreeBSD Open Source Project. *Information Systems Journal* **11**, 321–336.

Jørgensen, N. (2001b), Results of the FreeBSD Survey. Mimeo. (webhotel.ruc.dk/nielsj/research/freebsd/freebsd.html)

Kalai, E. (1977), Nonsymmetric Nash Solutions and Replications of 2-Person Bargaining, *International Journal of Game Theory* **6**, 129–133.

Kandori, M., G. Mailath und R. Rob (1993), Learning, Mutation, and Long-Run Equilibria in Games, *Econometrica* **61**, 29–56.

Katz, M. und C. Shapiro (1985), Network Externalities, Competition, and Compatibility. *American Economic Review* **75**, 424–440.

Katz, M. und C. Shapiro (1994), Systems Competition and Network Effects, *Journal of Economic Perspectives* **8**, 93–115.

Kelty, C. (2001), Free Software/Free Science, *First Monday* **6**.

Kernighan, B. und B. Pike (1984), *The UNIX Programming Environment*, Englewood Cliffs, Prentice Hall.

Krishnamurthy, S. (2002), Cave or Community? An Empirical Examination of 100 Mature Open Source Projects, *First Monday* **7**.

King, R. (2002), Linux Market Share Within Web Server Sector to Grow, *thewhir.com*, 7. Januar 2002.

Klemperer, P. (1995), Competition when Consumers have Switching Costs: An Overview with Applications to Industrial Organization, Macroeconomics, and International Trade, *Review of Economic Studies* **62**, 515–539.

Koch, S. und G. Schneider (2002), Effort, Co-operation and Co-ordination in an Open Source Software Project: GNOME, *Information Systems Journal* **12**, 27–42.

Kogut, B. und A. Metiu (2001), Open-Source Software Development and Distributed Innovation, *Oxford Review of Economic Policy* **17**, 248–264.

Lakhani, K. und E. von Hippel (2003), How Open Source Software Works: "Free" User-to-User Assistance, *Research Policy* **32**, 923–943.

Lakhani, K., R. Wolf, J. Bates und C. DiBona (2002), The Boston Consulting Group Hacker Survey, Präsentationsunterlagen, The Boston Consulting Group, Boston.

Lakhani, K. und R. Wolf (2005), Why Hackers do what they do: Understanding Motivation and Effort in Free/Open Source Software Projects. In J. Feller, B. Fitzgerald, S. Hissam und K. Lakhani (Hrsg.), *Perspectives on Free and Open Source Software*, Cambridge, MIT Press.

Lee, G. und R. Cole (2003), From a Firm-Based to a Community-Based Model of Knowledge Creation: The Case of the Linux Kernel Development, *Organization Science* **14**, 633–649.

Lee, S., N. Moisa und M. Weiss (2004), Conditions for Open Source as a Signalling Device, Mimeo, Johann-Wolfgang-Goethe-Universität, Abteilung Finanzen, Frankfurt.

Leppämäki, M. und M. Mustonen (2004), Signaling and Screening with Open Source Programming, Arbeitspapier W-377, Helsinki School of Economics.

Lerner, J. und J. Tirole (2002), Some Simple Economics of Open Source, *Journal of Industrial Economics* **50**, 197–234.

Matros, A. (2004), Virtuous versus Spiteful Behavior in a Public Good Game, Mimeo, Department of Economics, University of Pittsburgh.

Matutes, C. und P. Regibeau (1988), Mix and Match: Product Compatibility without Network Externalities, *RAND Journal of Economics* **19**, 221–234.

McConnell, S. (1999), Open-Source Methodology: Ready for Prime Time?, *IEEE Software* **16**, 6–8.

Mertens, P., F. Bodendorf, W. König, A. Picot, M. Schumann und T. Hess (2004), *Grundzüge der Wirtschaftsinformatik*, 9. Auflage, Berlin, Springer.

Misanec, M. (2001), Legale Aspekte von Open-Source-Software. In B. Marx (Hrsg.), *Linux Manager Guide*, Nürnberg, SuSE-Press, 43–66.

Mockus, A., R. Fielding und J. Herbsleb (2000), A Case Study of Open Source Software Development: The Apache Server. In *Proceedings of 22. International Conference on Software Engineering*, New York, ACM Press, 263–272.

Moody, G. (2001), *Rebel Code*, Cambridge, Perseus Publishing.

Moon, J. und L. Sproull (2002), Essence of Distributed Work: The Case of the Linux Kernel. In P. Hinds und S. Kiesler (Hrsg.), *Distributed Work*, Cambridge, MIT Press, 381–404.

Mortali, M. (2002), *Market Opportunity Analysis For Open Source Software*, Studie, Open Forum Europe, Surrey.

Mortali, M. (2003), *Market Perception Analysis of Open Source Software*, Studie, Open Forum Europe, Surrey.

Mustonen, M. (2005), When Does a Firm Support Substitute Open Source Programming?, *Journal of Economics and Management Strategy* **14**, 121–139.

Myatt, D. und C. Wallace (2002), Equlibrium Selection and Public Good Provision: The Development of Open-Source Software, *Oxford Review of Economic Policy* **18**, 446–461.

Nelson, P. (1970), Information und Consumer Behavior, *Journal of Political Economy* **78**, 311–329.

Nielsen//NetRatings (2004), 55 Millionen Europäer "googeln" im Netz, Pressemitteilung vom 4. März 2004, Nielsen//NetRatings Deutschland, Nürnberg.

OECD (2004), *Education at a Glance: OECD Indicators 2004*, Organisation for Economic Co-operation and Development, Paris.

O'Reilly, T. (2001), Shuttle Diplomacy Between Allchin and Stallman, O'Reilly Inc, Sebastopol. (www.oreillynet.com/cs/user/view/wlg/122)

PDS (2001), Internet, Stellungnahme der PDS-Abgeordneten im Bundestag, (archiv2001.pds-online.de/mdb/mdb/dokumente/themen/?zid=93)

Perens, B. (1999), The Open Source Definition. In C. DiBona, S. Ockman und M. Stone (Hrsg.), *Open Sources: Voices from the Open Source Revolution*, Sebastopol, O'Reilly, 171–188.

Pingle, M. und R. Day (1996), Modes of Economizing Behavior: Experimental Evidence, *Journal of Economic Behavior and Organization* **29**, 191–209.

President's Information Technology Advisory Committee (2000), Developing Open Source Software To Advance High End Computing, Report to the President. (www.hpcc.gov/pubs/pitac/pres-oss-11sep00.pdf)

Prüfer, J. (2004), Why do Developers and Firms Contribute to the Production of Open Source Software?, Mimeo, Fakultät für Volkswirtschaftslehre, Johann-Wolfgang-Goethe-Universität, Frankfurt.

QA Systems (2003), Java IDE Market Share Survey, QA Systems, Driebergen. (www.qa-systems.com/products/qstudioforjava/ide_marketshare.html)

Raymond, E. (1996), *The New Hacker's Dictionary*, 3. Auflage, Cambridge, MIT Press.

Raymond, E. (1999), Shut Up And Show Them The Code, Mimeo. (www.catb.org/~esr/writings/shut-up-and-show-them.html)

Raymond, E. (2001), *The Cathedral and the Bazaar*, Überarbeitete Auflage, Sebastopol, O'Reilly.

Richter, S. (2003), Patente auf Software?, *Die Zeit*, Jahrgang 2003, Nr. 30.

Robles, G., H. Schneider, I. Tretkowski und N. Weber (2001), Who Is Doing It? A Research on Libre Software Developers, Mimeo, Fachgebiet für Informatik und Gesellschaft, Technische Universität Berlin.

Robles, G. (2004), A Software Engineering Approach to Libre Software. In R. Gehring und B. Lutterbeck (Hrsg.), *Open Source Jahrbuch 2004: Zwischen Softwareentwicklung und Gesellschaftsmodell*, Berlin, Lehmanns Media, 193–208.

Rochet, J.-C. und J. Tirole (2002), Cooperation among Competitors: Some Economis of Payment Card Associations, *RAND Journal of Economics* **33**, 549–570.

Rochet, J.-C. und J. Tirole (2003), Platform Competition in Two-Sided Markets, *Journal of the European Economic Association* **1**, 990–1029.

Rohlfs, J. (1974), A Theory of Interdependent Demand for a Communication Service, *Bell Journal of Economics* **5**, 16–37.

Royce, W. (1970), Managing the Development of Large Software Systems. In *Proceedings IEEE WESCON*, IEEE Computer Society, Washington, 1–9.

Ryan, R. und E. Deci (2000), Intrinsic and Extrinsic Motivations: Classic Definitions and New Directions, *Contemporary Educational Psychology* **25**, 54–67.

Schlag, K. (1998), Justifying Imitation, Mimeo, Economic Theory III, Universität Bonn.

Schmidt, K. und M. Schnitzer (2003), Public Subsidies for Open Source? Some Economic Policy Issues of the Software Market, *Harvard Journal of Law and Technology* **16**, 473–505.

Selten, R. (1991), Evolution, Learning, and Economic Behavior. *Games and Economic Behavior* **3**, 3–24.

Shankland, S. (2003), SuSE Linux gets Security Credentials, *news.com*, 5. August 2003.

Shapiro, C. und H. Varian (1999), *Information Rules: A Strategic Guide to the Network Economy*, Boston, Harvard Business School Press.

Shy, O. (2001), *The Economics of Network Industries*, Cambridge, Cambridge University Press.

Smith, B. (2002), The Future of Software: Enabling the Marketplace to Decide. In R. Hahn (Hrsg.), *Government Policy toward Open Source Software*, Washington, Brookings Institution Press, 69–85.

SPD und Bündnis 90/Die Grünen (2001), Bundestags-Drucksache 14/5246, Deutscher Bundestag, Berlin.

Spence, M. (1973), Job Market Signaling, *Quarterly Journal of Economics* **87**, 355–374.

Stallman, R. (1999), The GNU Operating System and the Free Software Movement. In C. DiBona, S. Ockman und M. Stone (Hrsg.), *Open Sources: Voices from the Open Source Revolution*, Sebastopol, O'Reilly, 53–70.

Software Magazine (2004), The 2004 Software 500, King Content Co., Newton. (www.softwaremag.com/SW500/)

Thibodeau, P. (2004), Sidebar: Sun, OpenOffice Plan to Make Desktop Apps More Like Office, *computerworld.com*, 8. November 2004.

Torvalds, L. und D. Diamond(2001), *Just for Fun: The Story of an Accidental Revolutionary*, New York, HarperCollins.

Unilog (2003), *Client Studie der Landeshauptstadt München: Kurzfassung des Abschlussberichts inklusive Nachtrag*, Studie, Unilog Integrata Unternehmensberatung, München.

van Damme, E. und J. Weibull (2002), Evolution in Games with Endogenous Mistake Probabilities, *Journal of Economic Theory* **106**, 296–315.

Varian, H. (1998), Markets for Information Goods, Mimeo, School of Information Management and Systems, University of California at Berkeley.

von Hippel, E. (1998), Economics of Product Development by Users: The Impact of "Sticky" Local Information, *Management Science* **44**, 629-?644.

von Hippel, E. und G. von Krogh (2003), Open Source Software and the "Private-Collective" Innovation Model: Issues for Organization Science, *Organization Science* **14**, 209–223.

Wagner, M. (2000), Google Bets The Farm On Linux, *internetweek.com*, 1. Juni 2000.

Wagner, M. (2001), Google Defies Dotcom Downturn, *internetweek.com*, 27. April 2001.

Wheeler, D. (2005), Why Open Source Software / Free Software (OSS/FS, FLOSS, or FOSS)? Look at the Numbers!, Private Studie. `(www.dwheeler.com/oss_fs_why.html)`

Wichmann, T. (2002), Free/Libre and Open Source Software: Survey and Study, Part 1: Use of Open Source Software in Firms and Public Institutions, Berlecon Research GmbH, Berlin.

Wilcox, J. (2000), IBM to spend $ 1 Billion on Linux in 2001, *news.com*, 12. Dezember 2000.

Wong, W. und S. Junnarkar (1999), Net Messaging Standards War Brewing?, *news.com*, 23. Juli 1999.

Yankee Group (2004), Linux Not A Low-Cost Alternative to Unix and Windows for Large Enterprises, Pressemitteilung vom 5. April 2004, Yankee Group Research Inc., Boston.

Yasaki, Y. und R. Murakami (2003), Network Effects in the Japanese Word-Processing Software Market, Mimeo, Research Center for Advanced Science and Technology, University of Tokyo.

Young, H. (1993), The Evolution of Conventions, *Econometrica* **61**, 57–84.

Young, H. (1998), *Individual Strategy and Social Structure*, Princeton, Princeton University Press.

YOUNG, R. (1999), Giving It Away: How Red Hat Software Stumbled Across a New Economic Model and Helped Improve an Industry. In C. DiBona, S. Ockman und M. Stone (Hrsg.), *Open Sources: Voices from the Open Source Revolution*, Sebastopol, O'Reilly, 113–125.

Ehrenwörtliche Erklärung

Hiermit erkläre ich, die vorliegende Dissertation selbständig angefertigt und alle benutzten Hilfsmittel vollständig und deutlich angegeben zu haben.

Mannheim, den 13. Februar 2006

Jörg Gutsche

Lebenslauf

Name	Jörg Joachim Gutsche
Geburtstag und -ort	24. September 1973, Hannover

Ausbildung

08/2000 – 02/2006	Promotion in Volkswirtschaftslehre an der Universität Mannheim über das Thema „Ökonomische Analyse offener Software“, Nebenfach Marketing
10/1994 – 03/2000	Studium des Wirtschaftsingenieurwesens, Fachrichtung „Informatik und Operations Research“ an der Universität Karlsruhe (TH)
05/1993	Abitur am Gymnasium Bad Nenndorf, Leistungsfächer Mathematik und Physik

Berufliche Tätigkeiten und Praktika

08/2000 – 03/2006	Wissenschaftlicher Mitarbeiter am Lehrstuhl von Prof. Dr. Klaus Conrad an der Universität Mannheim
02/2000 – 07/2000	Berater bei The Boston Consulting Group, Düsseldorf
06/1999 – 08/1999	Praktikant bei The Boston Consulting Group, Düsseldorf
11/1997 – 12/1997	Praktikant bei MANAGEMENT consult, Mannheim
08/1997 – 10/1997	Praktikant bei McKinsey & Company, München