Janne Hemilä

HOW TO UTILIZE OPEN SOURCE SOFTWARE IN THE COMMERCIAL ENVIRONMENT

Aalto School of Science

Instructor and supervisor: Professor Ilkka Kauranen

Abstract

AALTO UNIVERSITY SCHOOL OF SCIENCE PO Box 12100, FI-00076 AALTO http://www.aalto.fi	ABSTRACT OF THE MASTER'S THESIS
AUTHOR: Janne Hemilä	
TITLE: How to utilize open source software in the commercial environment	
SCHOOL: School of Science	
DEPARTMENT: Department of industrial engineering and management	
PROFESSORSHIP: Strategic management	CODE: TU-91
SUPERVISOR: Professor Ilkka Kauranen INSTRUCTOR: Professor Ilkka Kauranen	

My experience working with open source software exposed a lack of comprehensive, easily graspable, introductory articles suitable for non-technical readers. The objective of this study is to provide the reader with a comprehensive understanding on how to utilize open source software in a commercial environment.

Through a literature review this study identifies common patterns among open source projects and related companies. Patterns have been organized into four identified domains: legal, social, technological and business. Real life examples of the patterns are provided to assist understanding.

In conclusion, this thesis argues that open source can be utilized to build successful commercial operations. Open source can be used to improve software development, software quality, to gain feedback, to expand the user base, to influence the direction of technological progress and to benefit from innovations, which would otherwise be hard to monetize. However, open source also introduces serious drawbacks and risks regarding commercial operations, including low appropriability, market destruction and loss of control of the software development.

DATE: Mar 23, 2015	LANGUAGE: English	NUMBER OF PAGES: 88

KEYWORDS: Open source, open source software, business models, software development, business development

AALTO-YLIOPISTO PERUSTIETEIDEN KORKEAKOULU PL 12100, 00076 Aalto http://www.aalto.fi	DIPLOMITYÖN TIIVISTELMÄ
TEKIJÄ: Janne Hemilä	
TYÖN NIMI: Avoimen lähdekoodin ohjelmistojen soveltaminen liiketoiminnassa	
KORKEAKOULU: Perustieteiden korkeakoulu	
LAITOS: Tuotantotalouden laitos	
PROFESSUURI: Strateginen johtaminen	KOODI: TU-91
TYÖN VALVOJA: Professori Ilkka Kauranen TYÖN OHJAAJA: Professori Ilkka Kauranen	

Pitkäaikainen työskentelyni avoimen lähdekoodin parissa on osoittanut puutteen kattavalle, helposti sisäistettävälle johdantoartikkeleille, jotka soveltuisivat myös maallikoille. Tämän tutkimuksen tavoitteena on tarjota lukijalle kattava ymmärrys avoimen lähdekoodin ohjelmistojen käyttämisestä kaupallisessa liiketoiminnassa.

Tutkimus esittelee kirjallisuudesta yleisiä säännönmukaisuuksia avoimen lähdekoodin projektien ja yritysten toiminnasta. Nämä säännönmukaisuudet on jaoteltu neljään tunnistettuun osa-alueeseen: lakitekniikka, sosiaalinen ympäristö, teknologia ja liiketoiminta. Ymmärtämisen helpottamiseksi tutkimukseen on kerätty esimerkkejä kaupallisista yrityksistä.

Johtopäätöksenä tämä tutkimus osoittaa, että avointa lähdekoodia voidaan käyttää menestyksekkään liiketoiminnan kehittämiseen. Avointa lähdekoodia voidaan hyödyntää ohjelmiston kehittämiseen, laadun parantamiseen, käyttäjäpalautteen keräämiseen, käyttäjämäärien kasvattamiseen, teknologisen kehityksen ohjaamiseen, tai sellaisien innovaatioiden hyödyntämiseen, joiden tuotteistus olisi muuten hankalaa tai mahdotonta. Avoimella lähdekoodilla on kuitenkin myös kauaskantoisia haittapuolia, kuten alentunut liikevaihto, lisensointimahdollisuuksien häviäminen, sekä ohjausvallan menetys ohjelmiston kehityksessä.

PÄIVÄMÄÄRÄ: Mar 23, 2015	KIELI: Englanti	SIVUMÄÄRÄ: 88

AVAINSANAT: Avoin lähdekoodi, liiketoimintamallit, ohjelmistokehitys, liiketoiminnan kehittäminen

Preface

Over my years as a software developer, both as a consultant and as an entrepreneur, I have come to understand the ubiquitous nature of open source software. Open source is not used only as a way to lower costs, but because open source solutions are often, all things considered, the only technically sensible alternative for a given platform, framework or programming language.

Without open source, there would be no Internet, no Facebook, no Twitter, no OS X, no iPhone, no GPS-navigators, no Amazon's server clouds, no Android phones nor many other technological marvels of the day. Despite its omnipresence and continuous growth, I have witnessed the concept of open source elude even otherwise savvy individuals. With an increasing weight of computing and data systems in almost every industry, better understanding of the fundamentals is required from even non-technical personnel.

I would like to express my deepest gratitude to my supervisor Prof. Ilkka Kauranen for guidance and support on how to be a better scholar. Also I would like to give my thanks to Michael Przybilski for the initial sparks of inspiration on how to turn the theme into a thesis. Also I'd like to acknowledge the continuous help of Koho Sales personnel, whose support made finishing this thesis possible.

Espoo Mar 23, 2015

Janne Hemilä

Term definitions

API
Application programming interface. Specifies how an application can exchange data with another application.

BSD
Berkley software distribution. An open source version of Unix.

Compiled code (binary code)
Source code that is processed and turned into computer-readable, binary format. Decompiling compiled code to source code is possible, but often unfeasible and produces poor quality source code. In practice, it is impossible to edit or study an application without access to its source code.

Compiler
Application that turns human-friendly source code into compiled code.

Coopetition
Cooperative competition. A scenario, where direct competitors sharing an ecosystem compete together against an incumbent ecosystem. By gaining marketshare for their ecosystem, they benefit more, than by gaining business from their direct competitors.

Forking
Taking a copy of a software project's source code and developing it separately. As a result, there are two versions of the software, the original and the fork. Also known as branching.

GUI
Graphical user interface. As opposed to the text-based command line interface (CLI), GUI provides a user with visual elements like buttons and windows.

I/O
Input/output. Any operations where a computer reads or writes to a hard disk or memory.

IDE
Integrated development environment. A text editor geared towards code development with additional, integrated features to ease the task, such as compilers, code analysis tools and version control.

IP
Intellectual property is a legal term that refers to creations of the mind. Examples of intellectual property include music, literature, and other artistic works; discoveries and inventions; and words, phrases, symbols, and designs. Under intellectual property laws, owners of intellectual property are granted certain exclusive rights.[1]

Regression
A bug in a new software version that breaks down existing, working features.

Open source community
Developers developing open source software with or without direct financial gain.

Open source license
A software license, which has been studied by the Open Source Initiative, and deemed to fulfill specified requirements.

Open source software
Software licensed under an open source licenses

Software license
Document specifying the privileges and restrictions the copyright owner grants to others using the product

Source Code
Human readable and editable code. Written by a human.

UI
User interface. Any software or hardware that allows a user to communicate with the computer. Modern user interface consists of a graphical user interface, with mouse and keyboard inputs.

[1] http://en.wikipedia.org/wiki/Intellectual_property

Table of Contents

1 Introduction

Every passing year, along with the pervasion of the Internet and software technology, increasing numbers of companies and professionals find themselves practically working in software industry, directly or indirectly. Albeit not as programmers, but as users and prototypers of new kinds of information systems and often closely co-operating with software vendors.

As technology has propagated through the business world, a new kind of software development scene has appeared and is gaining ground. Existence and nature of open source software, as it is called, can initially seem perplexing to many due to licensing quirks, communities and shared IP. For commercial companies, the core feature of open source, seemingly free access for users and competitors to product and its source code, is clearly against the incumbent paradigms of proprietary IP-control. Nevertheless, open source software has cut into company revenues and is still gaining momentum.

There is a lot of literature about open source and it's steady march of success. However, most of this literature is written by and for technical people or academics. For business-oriented professionals, little literature exists that would help them understand the concept of open source and how it can be used in for-profit business. Most of the more business-oriented material available tends to be extensively simplified.

To survive the threat of open source and to turn it into working products, happy customers and profit may require a major paradigm shift as well as reevaluating the whole business model and processes. Lack of knowledge of existing commercial open source successes and failures and poor of understanding of the concept and environment hinder the spread and adaptation of the open source paradigm.

Open source methodology is also applicable to hardware development, but this aspect is not covered in this study. Open source hardware development has recently gained more momentum, but is still in its infancy and material of its business applications is sparse.

Definition of open source

Open source is a licensing scheme for software. To qualify as an open source license, a software license must be approved by the Open Source Initiative. Open Source Initiative requires the license to provide the end user with certain privileges regarding changing and distributing the software, typically prohibited by commercial licenses. The license is the foundation of the entire open source movement, as it allows for novel forms of cooperation and development.

Social, economic and ethical issues

Around open source exists a plethora of social movements, ethical issues and major macro-economical repercussions. Open source contributors are often driven by philosophical ideologies and open source is easily seen as kind of a Rorschach inkblot test, where one can see whatever they desire, be it proof of superiority for libertarianism or communism or some other ideology.

This study takes no stance in these issues, nor does it attempt to frame open source as an ethical maxim, instead concentrating on the pragmatic business side of the phenomenon.

Target audience

This study is targeted to business-oriented readers and does not assume significant technical knowledge outside of relatively common concepts. Necessary details of open source and other technical issues are provided in sufficient level for understanding the contents.

This is not a how-to manual for a profitable business, as open source is not an isolated component, product or business model, that could be taken as is. Nor will this study offer a comprehensive map of open source, with mathematically and logically sound term definitions, frameworks or formulas.

1.1 Objective

The objective of this study is to provide the reader with a comprehensive understanding on how to utilize open source software in a commercial environment. This knowledge is organized into a manageable, well-scoped handbook. Summarized real life examples are provided to ease the understanding of complex scenarios.

1.2 Methodology

This study is carried out as a literature review, based on scientific articles and equivalent material, such as books.

- Initially 150 articles were selected through Google Scholar service based mainly on combinations of keywords “open source”, “business model”, “services” and “community”.
- Initial articles were filtered for non-relevant or otherwise questionable or unreliable content.
- Selected articles were processed into organized notes and citations, which were then used to form the structure of the study.
- Multiple news sites (e.g. slashdot.com, bbc.com, technewsworld.com) were searched for related news and other material.
- Notes, citations and other material were constructed into this study.

1.3 Structure: four domains

Open source brings about a host of related effects not directly related to the technology itself. In business environment, effects of open source phenomenon can be roughly divided into four separate domains: Social, legal, technical and business. This explicit division is based on similar, inexplicit structures in the literature, as well as on classification of common patterns discovered during the research. The concept of this domain division is present in the literature. However, it is not brought to its logical conclusion, but instead used as a general guideline.

Social
Open source development siphons its life force from the community. The community presents a plethora of unique social issues and opportunities.

Legal
The whole concept of open source bases its existence on the existence and nature of open source licenses. These licenses are complex legal documents that have profound consequences for the licensed software.

Technical
Open source development methodology is specifically suitable for developing highly complex and technical software, and the whole phenomenon, legislation and experiences are based around this. Success stories of non-software open source development are quite rare.

Business
Utilizing open source software often has strong business repercussions regarding pricing, sales processes, intellectual property -policies and competition.

2 Open source and software industry

"Every industry that becomes digital eventually becomes free"
– Chris Anderson, Editor-in-chief Wired Magazine

Open source can be seen as a new way to develop software, which threatens incumbent software business (46, s. 310). As a challenge, it is not only about efficiency, but a more fundamental and structural, going back to the basic motivations, economics, market structure and philosophy of the institutions that develop, market and use software.

Academic backgrounds

Though often considered a new phenomenon, sharing software for free is in itself an old tradition, dating back to the "free software community" far before Harvard professor Richard Stallman's involvement in 1971. Back then, Stallman had problems with a faulty printer driver. He was a skilled programmer and could have fixed the problem himself, but he did not have access to driver's proprietary source code.

First users of computers in the 50s were mostly researchers and ownership of program code was not considered important. Sharing the source was required for other participants to make changes. This was a major distinction to later times, as software was not developed by for-profit companies, but exclusively by the users themselves.

In 1983, Richard Stallman launched the GNU project, aiming to create a completely free operating system without license constraints. Final trigger for him was a faulty printer driver Stallman could not fix due the undisclosed source code. The GNU project has produced several highly used programs.

Stallman argued that all software should be "free software," with source code that can be read, modified and redistributed (61, s. 7), a point of view more akin to science and scientific community, than business.

However, perceptions of free software are often more polarized than reality warrants, as even Stallman was not hostile towards commercial concerns. Instead, software should be considered more as a professional service, than intellectual property and appropriation should be measured according to development activity, not distribution (25, s. 46).

The GNU operating system project itself did not take flight due to the lack of operating system core, the kernel, until 1991, when Linux filled the spot. After the GNU/Linux started to gain momentum, it also spurred the open source community, which also greatly benefited from the appearance of the Internet, growing developer pool and improving tools.

The recent late boost in popularity can be seen as a confluence of three factors in 90s (61, s. 6):

1. Demand for an inexpensive Unix implementation.
2. Movement rejecting the idea of software ownership and profitability.
3. Emergence of Internet and collaboration it enables, and tools its use requires.

Ownership and liability structure

It is not feasible to consider open source as a technological advancement, but more as a social and legal one (46, s. 310). Peter Drucker considers the emergence of open source to be more akin to the emergence of limited liability companies, trade unions or newspapers, which transform existing institutions in ways beforehand unimaginable and leave them unrecognizable (60, s. 17). Though naturally in a much smaller magnitude, as the appearance of limited liability companies changed the face of business completely, but open source is still a limited factor even in software industry. Like other hype terms like big data, open source is means to an end, not a standalone institution. The end being efficient development and profitable businesses.

Requirements

To be considered open source, software has to meet some requirements that can be summarized thusly (60, s. 6)

1. Source code must be distributed with the software or otherwise made available for no more than the cost of distribution
2. Anyone may redistribute the software for free, without royalties or licensing fees to the author
3. Anyone may modify the software or derive other software from it, and then distribute the modified software under the same terms.

In short, open source is a licensing scheme for source code. When a program is released as open source, its human-readable source code is included along with a license permitting the end user to modify and redistribute the program. Usually in proprietary settings only the compiled, non-modifiable, version of the code is distributed due the obvious risk of competitor copying it.

Open source licenses often restrict the use of the modified program to varying degrees (within boundaries specified by the open source initiative).A typical requirement is that any software derived from the open source software must also be distributed with the same license. This institutes a hurdle for for-profit companies, who cannot simply take an open source application and sell it as their own.

Distributed community work (crowd-sourcing)

Due the licensing scheme, open source enables multiple individual users to work on the software, forming the open source community. Community is the lifeblood of open source, as it drastically helps in development, testing and debugging of popular open source software.

Not a business model

Open source is not a business model itself in any meaningful sense of the word, though often referenced as such. Open source is effectively a licensing and development scheme that companies can utilize however they choose to, and position themselves in the environment wherever they fit. An "open source –company" can be a 5-man sweatshop developing a single product, or a billion dollar service powerhouse, both building their own business model incorporating open source in some form.

1.4 Information economics

Compared to almost every other industry in existence, software industry follows it's own natural laws and peculiarities. Success of open source software originates from these features. In this chapter are listed several major features that dictate the nature of software industry.

Network externalities

Software is a network good, highly prone to network externalities (30, s. 3). Perceived value of an application is often extremely dependent on how many other users the application has (2, s. 4). Enabling a two-way communication between networked agents provides value based on the size of the network, providing "synchronization value" to the users (41).

Magnitude of the network externality effect varies case by case, with most ludicrous examples being found in communication applications, such as Facebook, but also in platforms such as operating systems. The overwhelming reign of Windows was originally dependent on other applications, such as Word, not being compatible with other operating systems, Windows being the most attractive with most diffusion (2, s. 5)(7). Also for the complementary service providers, the platform with the largest installed base is the most valuable.

Taking the network externalities into account is often extremely important, especially in the early stages of the project, with each adaptation decision shaping the value of the network (2, s. 5). Depending on the switching costs, the one with most users may attain dominant position, even with an inferior product.

Basic research and platforms

Throughout the history, private, for-profit companies have been inefficient in basic research, be it in physics, biology or in any other field. In software, there are constructs similar to basic research. For example web servers and programming languages, all of which are absolutely vital for development of software, are rarely commercial successes (30, s. 7).

However, open source projects have turned out to be a competent alternative for producing collectively useful components (30, s. 6). Motivations of open source developers are also aligned with those of scientists.

Prototyping

Compared to almost any other industry, altering an existing software installation afterwards is trivial. Especially in heavy industry and construction, no such concept exists to begin with. This means that a provider can deliver a known faulty or restricted product to a customer and afterwards quickly fix it with relatively little additional cost. Though delivering faulty products does not seem as a good practice, it is often necessary, as some fixes and features can only be finished based on customer feedback. This prototyping procedure enables extremely fast development-adaptation cycles, where customer feedback can be gathered, incorporated into the product and then delivered back to the customer in a matter of days, or even hours.

Uniformity of technology

Software development tools are practically global. Regardless of the geographical or ethnic background or language spoken, programming languages and tools stay the same. As a result, open source projects can incorporate code from all over the world.

Reliance human capital

Software development itself is extremely light on resources, relying almost exclusively on human capital. With second-hand laptops and desktops easily available and most development tools available for free, upfront costs of starting a software project are essentially zero.

Developers are global and plentiful

Lately, many countries all over the world have greatly increased the rate of programming education[2]. For example India has invested significantly in education, and by 2017, is predicted to overtake US in the number of software developers. In 2013 there were estimated 18 million software developers worldwide and the number is expected to rise to 26.4 million by 2019. Unlike many other specialists that are tied to locations, languages and tools, these developers are relatively uniform and can work in same projects over the world. Any project accessible through the Internet can be developed by anyone.

Technical progress

Along with the number of developers increasing, the quality of software development tools, such as editors and languages, has also steadily increased. A programmer in the 70s needed to learn cryptic, hard to learn, unintuitive languages with poor text-based editors. Newer programming languages such as PHP or Ruby that are syntactically intuitive and easy to use and learn even by children with little effort. These improvements are backed by cheaper and more powerful hardware, with a modern cellphone having magnitudes more capacity than a top class PC decades ago.

[2] http://www.computerworld.com/s/article/9240676/India_to_overtake_U.S._on_number_of_developers_by_2017

With improved IDE (integrated development environment) code editors, user-friendly languages and cheap hardware, learning programming is not restricted to selected few, but to anyone with an Internet connection.

In the long run, everything will be free

The nature and success of open source and economics somewhat guarantee, that should a high-demand, commercial product either lack in features or command too high a price, eventually there will be an open source project attempting to recreate the same application as open source. This has happened to innumerable commercial products over time, such as Unix, Photoshop and Microsoft Office. Initially, open source competitors will naturally be inadequate, but over time, should the demand persist, they tend to gain momentum and features, eventually competing with the incumbent for same client base.

Though unstoppable at best, competitive effects of open source alternatives vary and should not be overstated. Often developer motivations are satisfied with a lower quality product than the commercial version, development rate dropping as more and more features are considered good enough. Commercial companies can also often mitigate the motivations by simple licensing and pricing schemes.

No rivalry of consumption

Software is freely copied and delivered with effectively zero cost. Giving away information does not preclude consumption by the donator (49, s. 5). The difficulty to appropriate this consumption is responsible for the high death rate among software ventures (30, s. 2).

Piracy

With ubiquitous Internet, delivering and downloading software illegally ("pirating") has often become easier than actually buying it. Despite efforts of commercial companies to threaten and jail pirates, typically adolescent males, piracy has not declined. Attempts to close torrent sites such as the Pirate Bay have mostly been exercises in futility, with pirates finding two new delivery routes for every one closed.

Some companies, such as Microsoft, have accepted piracy, and adapted accordingly. For example, Microsoft offers security updates for all Windows installations, including ones without a proper license. With tools like the TOR-router, which allows users to browse the Internet with complete anonymity, piracy is here to stay.

Evolution over revolution

Artifacts of software development are not measurable in the number of lines of code, but in the innovations, design features and the architecture the code implements. Lines of code can be generated as pay-per-hour. Innovations however are generated more by chance, given required conditions. By utilizing a community of developers, a project also gains specialized features not needed or planned by the initial developers, but which may turn out to be a key issue for a wider audience.

In companies, value lies in combining in-house resources and expertise with the inventive activity in the open source community, though value appropriation is not straightforward and requires other kinds of methods than in the purely proprietary sector.

Lack of costs with changes is the key factor enabling the iterative, distributed development approach of software. Let the community picker, fight, argue and develop, and see what sticks (25, s. 41).

1.5 Open innovation

"Open innovation is fundamentally about operating in a world of abundant knowledge, where not all the smart people work for you, so you better go find them, connect to them, and build upon what they can do" – Henry Chesbough, Executive director, center of open innovation, Haas school of business, UC Berkley

The Cathedral and the Bazaar

In his book "The Cathedral and the Bazaar", Eric S. Raymond writes about the two major paradigms of software development and their natures.

In a centralized model, decisions and plans are made by a handful of specialized "wizards" working in isolation, avoiding mixed interests and releasing the software "when it is done", as if building a **cathedral**, where no mistakes are allowed.

In the other, distributed model, open community works on the project like a swarm of bees, with little restrictions, weak hierarchy, crossing interests, little planning, infighting, resembling more a babbling **bazaar** than a sophisticated development process. Distributed model often results in releasing alpha and beta versions early and frequently.

No cathedral can ever be built with the bazaar methodology. But still the Linux kernel and related communities (gnome, KDE, whatnot) have already established a strong position in the IT world and are evolving faster than any cathedral builder could ever imagine.

Although the comparison of building software and building buildings is easy to grasp, it is also a reason why many intuitively feel the distributed model paradoxical and impossible. Main reason is the drastically different nature and economics between software and real-world constructions. Whereas the main issue of building a cathedral is the implementation of the plan including logistics, material resources and work process, in the software world the implementation, or code compilation, is automatic, free and extremely fast. In this sense, the major issue in software development is making the plans and specifications, the source code. Changes in the plans are trivially easy to test, and major changes can be done afterwards. Naturally, major refactoring of a huge system is in no way trivial due the changes needed in other parts of the specifications, but it is still possible, unlike rebuilding foundations of a cathedral.

Company borders

For commercial companies, supporting innovation and innovation sharing across companies is often a major challenge (49, s. 10). Especially in direct competition, sharing of mutually benefiting knowledge is often hard to execute, due to fears of losing position. Incorporating third party, collective innovation is also not straightforward, due to for example common antipathies against commercial companies and individuals' fear of losing credit for their work.

In software industry, ubiquity of the Internet and presence of common tools and languages offers an exceptional stage for collective innovation. Open source development has been often described as the most successful case of collective invention (49, s. 11).

Collective innovation itself is not a new idea. Mid 19th century iron-making companies in Britain's Cleveland district willingly shared their innovations in furnace design for free (49, s. 2). Though this might seem economically puzzling, the situation was more complex than companies being generous. Most of their inventions were incremental innovations, which other participants would have themselves found out in time, and keeping one's innovations a secrets was not 100% guarantee of it staying a secret. Instead, collectively sharing information left everyone better off. Other such historical examples include the development of steam engines and flat panel displays.

Successful open innovation can be used for (63, s. 4):

- Generating innovations to be internally commercialized (the proprietary model)
- Building absorptive capacity and using that capacity to identify external innovations
- Generating innovations that generate returns through external commercialization (e.g. licensing patent portfolios)
- Generating intellectual property that does not produce direct economic benefit, but indirectly generates a return through spillovers or sale of related goods and products

Coopetition

Commercial motivations for sharing information can better be understood as a case of coopetition, where companies not only compete among themselves, but inside a wider ecosystem against other ecosystems. Though they might lose on their share of the pie, the pie itself might grow (49, s. 10).

Coopetition is a common pattern among open source companies. For example, dozens of commercial companies that distribute their own versions of GNU/Linux operating system have appeared. These companies often use and develop open source components used by their direct competitors, instead of developing their own proprietary code and keeping it a secret. This can be easily understood when considering the Linux ecosystem as a minor factor competing against Microsoft.

Absorptive capability

Common antagonist of successful incorporation of external innovation is the so-called "Not-invented-here"-syndrome. Oftentimes engineers and designers tend to scoff at external ideas in favor of their own (63, s. 5). For example Apple engineers in 1980s rejected ideas such as hand held computers, adopting the aforementioned phrase.

For a software company living with an open source community, or in an ecosystem with coopetitors, "not invented here"-attitude is a poison that has to be gotten rid of. Existence of external innovation, such as new technology, is of no value, should the firm lack capabilities for identifying the relevant technology and absorptive capability to incorporate it in it's own activities. This requires active scanning and internal political will (63, s. 3). There is direct evidence that in some industries, such as pharmacy, higher absorptive capacity directly translates into substantial gains in research and development productivity.

Appropriating internal innovation

Not every innovation can be easily monetized by packaging and selling. Instead, companies need to also look for other vectors of appropriation, such as licensing, patent pooling and sharing technology for free, to stimulate demand for complementaries (63, s. 3).

Maintaining external motivation

All external innovations are provided by some other person or entity, outside of organizational borders. Ensuring motivations for the given innovator to share knowledge in the future is essential (4, s. 19). In open source context the largest external entity is often the developer community.

In open source context, community developers often represent (and are) users (38, s. 199). As such, open source development process is an instance of more common user driven innovation, present in fields such as scientific instruments and machine tool industry. Often a solution developed by a single user can be incorporated into the product and become a general solution for a wider class of users.

Power of open source innovation is easily overstated due to idealism and survivorship bias, but its effectivity is highly context dependent and situational. Open source often has more power in imitation and less in innovation (4, s. 22).

3 Domains

3.1 Domain: Legal

Licensing is a core aspect of open source business and development. This chapter explains the role of licenses and introduces most common licenses and their differences.

3.1.1 Licensing and Code Ownership

As with all forms of intellectual property, the concept of ownership in software is hazier than with physical goods. When selling software, typically the ownership of the software does not change. Instead, buyer, or licensee, is given a permission to use the software with agreed limitations. For example, software may only be used in a specific organization, server or industry, or the maximum number of users may be limited. Typically licenses are used to restrict user privileges. It can be argued that the success of open source is partly due to contemporary inefficiencies of enforcing intellectual property rights in commercial setting (30, s. 4).

Instead of restricting, open source licenses tend to provide privileges, ensuring that any user of open source software is given proper access to the code and allowed to modify and edit it. This is a counter intuitive detail in the concept, as open source actually guarantees software to be used and accessed freely, by restricting potential intellectual property policies a commercial actor might enforce.

Legal issues regarding open source software development are often quite complex and convoluted. In 2003, Pamela Jones founded a legal blog “Groklaw”[3] as “a place where lawyers and geeks could explain things to each other and work together, so they'd understand each other's work better”[4].

Requirements for an open source license

“Open source” refers to a specific set of software licenses approved by the non-profit Open Source Initiative, OSI, which keeps a list[5] of licenses it considers to fill the open source criteria and, thus, can actually be called as “Open source license”. To be considered open source, a license must fill the following definition, summarized from the official definition.[6]

[3] http://www.groklaw.net/
[4] http://www.fsf.org/news/2007_free_software_awards
[5] http://opensource.org/licenses/alphabetical
[6] http://www.opensource.org/docs/osd

1. Free Redistribution
Every party must have the privilege to distribute the software without any fees.

2. Source Code
Distributed program must include the source code, and allow the redistribution of the source code. Source code must be in human readable and editable format.

3. Derived Works
The license must allow modifications and derived works and allow them to be distributed under the same terms.

4. Integrity of The Author's Source Code
The license may require derived works to carry a different name or version number from the original software. This provision protects the authors from gaining a negative reputation from the problems of changes that others have made to their work.

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 license must be able to be legally distributed along with the software itself.

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.

9. License Must Not Restrict Other Software
The license must not place restrictions on other software that is distributed along with the licensed software.

10. License Must Be Technology-Neutral
No provision of the license may be predicated on any individual technology or style of interface.

Since the open source definition was drafted in 1998, the Open Source Initiative has approved 54 licenses that meet the 10 requirements of an open source license. 15 of these open source licenses could be identified as firm specific and 14 as product specific. An average contributor will not and is not required to understand all nuances between these documents.

Open source is not public domain

A common misconception is that free and open source software is in the public domain. This happens simply because the idea of free software or Open Source is often considered confusing and people mistakenly describe these programs as public domain because that's the closest mental concept available. The programs, however, are clearly copyrighted and covered by a license and all except the owner of the software must abide by the license.

When it comes to truly public domain software, one can even re-license a public domain program, removing that version from the public domain, or one can remove the author's name and treat it as one’s own work. Open source licenses are often designed to allow for maximal freedom to the user, but little to no commercial use, as open source software often cannot be effectively sold.

It is fully possible to forfeit ownership and copyright of a code base, releasing it as public domain, without any license. However, this is rarely done and carries no benefits over licensing it with highly unrestrictive licenses.

Viral clause and General Public License (GPL)

A notable feature of many open source licenses is the way it restricts the licenses of any derived works, often by requiring derived works to be distributed with the original, viral license. This guarantees that any code that begins free remains free. This feature is often called Copyleft, a pun originated by Richard Stallman, denoting that the user still has his rights left, not taken. As the term “viral” often is associated with negative connotations, these licenses are often called reciprocal or restrictive.

If a company incorporates Copyleft source code in its products, it must provide the source code for these with the same Copyleft license, should it ever distribute the application. Just for emphasis, providing source code is only required, if a company distributes the application. Developing and using an application internally imposes no requirements. Still, GPL style licenses are often unattractive to commercial companies due to imposed limitations(15, s. 593).

Most common of the Copyleft licenses is the GNU General Public License, abbreviated GPL. Other licenses are often described as “GPL-style”, should they contain same viral restrictions. LGPL (GNU Lesser General Public License), a derivate of GPL, allows LGPL licensed software to be used as a module, without requiring the using software to be affected. GPL is the most common and well-known open source license in general[7]. For example Linux is licensed under GPL.

[7] https://www.blackducksoftware.com/resources/data/top-20-open-source-licenses

GPL versions

Over the years, GPL has evolved into several versions. Most used version is version 2, whereas most recent edition is version 3. Changes in version 3 are non-trivial and have sparked some discussion in the open source community. Main changes in version 3 address following issues[8]:

- International terminology, instead of using US legal concepts.
- Specific management of patents.
- Restrictions in consumer products, such as the digital video recorder TiVo, that take away consumer's ability to modify software.
- Digital rights management.
- Compatibility with other open source licenses.
- What kind of legal procedures will take place and by whom, should the license be violated.

Unrestrictive licenses

Contrasting to strict GPL, BSD (Berkley Software Distribution) style licenses are more permissive for commercial actors. BSD licenses allow software to be distributed quite freely, even as a component of proprietary software. However, BSD licenses require distributor to give credit for contributors of the software.

From a business perspective, BSD style licenses are attractive, as there are little to no restrictions on the future use and distribution (11). BSD style licenses are also ideal when promoting an application as a standard, e.g. as a communication protocol.

On the downside, there is little to encourage companies to share their own development back to the project. Though in a situation of coopetition, companies will still benefit from maintaining project's momentum. BSD-style licenses are often called “corporate” style. Other notable variants include Mozilla Public License (MPL), which adds provisions for protecting the company and community against patent issues. This is achieved by requiring a contributor to release any and all claims to patent rights incorporated in the source code. Apache license on the other hand gives a more detailed description on how a copyright notice should be included.

For example BSD-Unix variants such as Open-BSD are licensed under BSD-license.

Rewriting code

If a company wants to utilize existing open source software as a component in a proprietary product, it can rewrite the application, clearing all licensing issues and leaving the resulting code base fully in it's ownership. To be legally defensible, the rewritten version must be significantly different from the original. Often even large companies can consider an endeavor like this unfeasible.

[8] http://www.groklaw.net/article.php?story=20060118155841115

A more common case for rewriting code is when a company maintains a dual licensing scheme. In a situation like this, the company has two versions of the code: one is the open source version, where community contributions are included, and another a proprietary one, which is fully owned by the company. To incorporate contributions from the community, the company has to rewrite them to avoid diluting their ownership and risk losing the ownership of the application (54).

Rewritten code is immune to copyright notices, as copyright protection, unlike patent protection, only protects the expression, not the functionality. Rewriting and using patented code without a license to the patent is still a patent infringement.

Licenses are not tested

When it comes to legal structures, open source licenses have not been thoroughly tested in the court. Also detecting and proving that a company has incorporated open source code without adhering to the license can be an arduous and expensive task. Even GPL, most common of the licenses, has not been properly tested for enforceability (38, s. 3).

Even without a proven legal backing, open source licenses are relatively effective in protecting the community's work, as the whole community is willing to accept and internalize the license (49, s. 19). Still, a skeptic might ask, whether lack of legal confrontations lies in good behavior or well-hidden, or disregarded bad behavior.

As even GPL has not been extensively tested in the court, many less used, non-conforming licenses[9] have practically no guarantee of validity. Open Source Initiative keeps track on licenses and shares comments about them[10], stating that some licenses, such as PERL's Artistic License, should not be used due being “too clever” and vague.

Patents in the code

When code is licensed as open source, it is required to carry with it a grant of license to any patents concerning the software. Without such grants, any users of that open source software are vulnerable to legal litigations. It is possible for a company or individual to offer code into a project, and afterwards demand fee for patents in the code. Potential patent litigations are a deterrent for commercial actors, who are often hunted for patent lawsuits (37, s. 25).

The GPL specifies that if some piece of code is found to infringe a patent, thus restricting distribution of the code, then that code cannot be distributed at all. This clause is necessary to prevent “joint hijacking” by the patent owner and an open source co-conspirator, where latter could then receive exclusive license from the patent owner. These provisions have typically not been included in non-GPL licenses (38, s. 10)(37, s. 25).

[9] https://www.gnu.org/licenses/license-list.html#GPLIncompatibleLicenses
[10] http://opensource.com/law/13/1/which-open-source-software-license-should-i-use

Irrevocability

Once a company has licensed their software under any open source license and any single individual has downloaded that code under given open source license, this individual has unlimited permissions to distribute that version of the software under the same license, forever. When a version of software is open sourced, it will stay that way and cannot be undone, and if no community support can be gathered, significant deal of potential earnings may be wasted (61).

The licensor company is not required to distribute the code afterwards, nor required to share further results of their internal development as open source, as they own the copyright to the code. If company opens up proprietary software and gains no community support, it can usually just ignore the issue and carry on developing the product further internally. As most of the expertize still lies in the company itself, it is highly unlikely that a remarkable competitor appears, simply building on the soon-to-be-deprecated open source versions. In time, further development effectively deprecates older versions.

Threat of an irrevocable mistake can potentially be tuned down by modularizing software architecture and only opening some modules, where community support and need is more likely, and loss of potential revenues due to copying smaller.

3.1.2 Choosing a license

When either releasing an existing code base as open source or starting a new project, a choice must be made regarding which license to use. Along with practical issues such as code quality and project leadership, the choice of license will heavily influence the course of the project and it's attractiveness to potential contributors (27, s. 8). To make an informed choice, the licensor must assess a mixture of internal and external motivations, such as the company's business model, environment, industry, size and the architecture of the code base, intended audience and competition.

Often a license choice that is optimal for private companies may not be optimal for the community (38, s. 9). A common argument in favor of unrestrictive licenses is that permissiveness is what it takes to attract commercial software developers to write applications that enhance the value of the open source code. In particular in mature projects, when the energy of the initial contributors may be fading, the involvement of the commercial contributors may be critical to success.

Modifications

Open source licenses allow anyone to modify the source code, but are divided on whether or not these modifications should be made public or not when derived work is distributed further. Allowing modifications to be taken privately is invaluable to commercial actors, as they can invest internal resources in the project without fear of losing options for appropriation.

On the other hand, option to privatize modifications can be a deterrent for individual contributors, as they often dislike commercial companies using their code without contributing themselves (38, s. 9). Commercial companies have the option to copy the whole code base and begin selling their own, competing software, thus reducing potential markets for the open source alternative.

Various BSD-Unix versions have been used as components in Apple's OS X, Microsoft's Windows and Sun's Solaris. Still, this attention has not translated into development momentum, compared to the indirect GPL competitor, Linux. Reciprocal licenses such as GPL and LGPL prevent distribution of modified code without opening the source code, whereas BSD-style licenses allow this.

Ecosystem and complementarities

If an application is intended to operate along with other software, open source or proprietary, strongly reciprocal licenses cannot be used (38, s. 9). If open source project is only a stepping-stone in company's business model, with e.g. main income deriving from services and proprietary complementary software, utilizing a BSD-style license may encourage commercial users, thus increasing licensor's profit potential. If the field is saturated with restrictive licenses, utilizing one can make more sense than if potential complementaries were already licensed with unrestrictive licenses (38, s. 10).

A case in point is the choice of license by a company trying to get their software established as a standard. Despite an increased risk in hijacking, unrestrictive licenses are more feasible in such context (38). Especially in Internet/network related fields there often exist strong network externalities, where adaptation speed and scope can make or break a company, restrictive licenses have less value.

Licensing choices also give rise to network externalities among licensors. If existing projects in the field have restrictive licenses, a licensor may be likely to choose a restrictive license partly in the anticipation of future users combining the software.

Still, commercial interest is often not clear-cut. Linux enjoys a lot of interest and investment by major commercial players despite having restrictive license, whereas BSD versions have much less hype around them, even with their unrestrictive license. This may partly be due BSDs having forked into separate communities.

> *In early stages of the Internet, there were enormous differences between browsers regarding JavaScript that developers often had to effectively rewrite the same code for different browsers. This amounted to a significant load when developing interactive pages. Eventually W3C laid the groundwork for the now common Document Object Model that eventually grew into a common standard for JavaScript (11).*

Forking

Requirements of open source licenses state, that the software must be allowed to be distributed under the original license. Thus, there is nothing in the legal structure of the licenses itself that could directly alleviate the threat of forking. Instead, forking should be considered more of an economic choice; an unrestrictive license draws more commercial attention, which may lead to companies pressuring the community more, or even starting a fork to be developed with their internal resources.

Hijacking

It can be argued that unrestrictive licenses are particularly prone to hijacking by commercial companies. A company may include some proprietary code to the software, making the end result private. Though the end result may or may not be superior, the company disrupts the dynamics of the community by de facto privatizing the software. Though the original project will still exist under the original rules, there is a risk that the commercial version will confuse customers and thus dominate (38). Utilizing a restrictive license forces the potential hijacker to rewrite the entire code base, making it a costly endeavor.

Familiarity of open source community with the license

A licensor receives benefits in form of reduced transaction costs when he adapts a license the open source community is familiar with. Choosing a well-known license benefits from the community already knowing the implications the license has on development processes (38, s. 10). Most open source developers have no legal knowledge and thus rely on experience and hearsay on how a license works.

Restrictive licenses

Historically, highly restrictive licenses have been popular in certain situations: (37, s. 17)

- Mature projects. This pattern may reflect a “vintage effect”: it may have more common for older projects to employ licenses other than the GPL. Alternatively, this may reflect a “survival effect”.
- Projects that run under the POSIX family of operating systems.
- Applications geared towards end users.
- Projects aimed at systems administrators.
- Applications that are consumer oriented, e.g. desktop tools and games.
- Projects whose natural language is other than English, with the exception of Japanese.
- Projects that involve software developed in a corporate setting.
- When community appeal is weak, such as with an unknown licensor.

Restrictive licenses have been less popular in situations such as:

- Projects operating in commercial environments such as Microsoft Windows or Apple’s Cocoa.
- Applications aimed towards software developers.

- Community appeal is strong, such as with a well-known licensor.

3.1.3 Dual licensing

If a company has managed to keep a version of the code base undiluted and fully private, by for example rewriting open source contributions, an option for dual licensing arises. In this scheme, the community offers the open source version for free with a strongly reciprocal license, most often GPL, thus more easily gaining a strong community around the product (30, s. 16). However, the company still has the option to sell the license of the private version to any potential customers, who value the software itself, but cannot operate under the GPL license. Additionally, proprietary version may contain additional features, not present in the open source version (29, s. 1).

In practice, dual licensing is a form of price discrimination, where the product is spread and marketed for free, but commercial actors, ones with a higher cash flow, must pay for the usage. Permission to use software internally for zero cost, without requirements to disclose modifications, is more valuable than a money back guarantee; acquisition of an open source version is trivial for the customer, with no cash traffic required, compared to highly supervised trial licensing practices. Evidence implies that enforcing copyright is usually not needed, as corporate customers required to purchase a proprietary license tend to do so (54).

Utilizing a dual licensing scheme efficiently has some requirements for the project. First, user base must be sufficiently large. Copyleft license enables strong network effects, helping company to gain market share in users. Secondly, there must be an option for price discrimination, e.g. there must be a corporate customer, with enough money to pay for a license, and an alternative demand, that is not covered by the open source version, such as an embedded product.

There are some legal considerations in establishing and maintaining proper legal practices between company's private and open source activities. Sufficient education of employees, adequate documentation and procedures are required to ensure sufficient isolation and prevent issues such as ownership dilution.

> *Id Software has utilized dual licensing scheme around their game engine called "Unreal engine". In game development scheme there is a large base of amateurs developing games and game modifications as a hobby. These individuals are able to utilize Unreal engine for free in their projects, allowing the software to gain enormous user base. However from time to time, these free time developers end up founding an actual, commercial game studio. With experience in Unreal engine, they are willing to pay for the commercial license required for commercial distribution.*

3.2 Domain: Social

Open source allows all interested parties, such as other companies and individuals, to read and modify the source code. Often these developers are willing to contribute their changes back to the original version for free. When these pro-bono developers are helped to organize and coordinate their development efforts, they form a community, which in turn helps in development, testing, documentation and shares innovations and ideas.

These communities form the driving force of open source development. Nurturing and maintaining a vivid community is crucial for the success of an open source project. Gaining developer goodwill offers the company a larger base on more interested free workforce, ensuring better products, which amount to better customer offerings and better value propositions.

Growth of crowd-sourcing

With the growing trend of empowered and knowledgeable consumers, willing and able to participate more directly in the creation of complex products with limited manufacturer involvement, many companies have altered their traditional communication model to a more reciprocal one. This growing phenomenon is not limited to software industry; windsurfers have for a long time customized, developed and manufactured some of their own gear (20, s. 82). Crowd-funding services like Kickstarter are also riding the same wave of consumer empowerment. Open source development is simply an instance of organized crowd sourcing, enabled by the Internet, common tools, uniform technology and open source licenses.

Developers are users

For a person to design his own windsurfing gear, one has to be an experienced windsurfer. Likewise, for one to take part in developing software, one must know the relevant domain of that software. Meaningful code contribution in an open source project naturally requires some basic understanding of software programming, but more importantly, knowledge and expertize in the given field. Even with little to no programming skills, an innovative user may come up with ideas for improvements (30, s. 23).

As such, open source developers tend to be experienced users of the software they write and familiar with the requirements and desirable behavior (46, s. 330). This is often not the case in most commercial software companies, where the developers are mostly hired for their experience in development, not in the target field itself (8).

Appeal for a community

For a chance for the community to grow, three conditions must be met (20, s. 84):

- Users have sufficient incentive and ability to innovate.

- Users have an incentive to reveal their innovations and the means to do so, e.g. by forming their ideas into application code.
- User-led, free diffusion of innovations can compete with commercial production and distribution.

Projects with unsophisticated end users as the intended audience, such as desktop applications and games, often have a great number of end users. However, these users often lack technical capabilities and will to produce sufficiently high-class user interfaces to be used by non-technical users. Ego gratification and career incentives also have little power, if the target audience is not likely to look at the code.

Improved innovation

In software development, innovation often originates from users, not the producers (55).
By making frontier users into co-developers, a company can significantly speed up debugging, quality and feature development (30, s. 23). Open source developers are often driven by intrinsic motivations, such as fun, and are eager to test their brainpower against a problem.

Donated complements

Donated code is the most central benefit of an efficient community. Companies with insufficient development capacity can leverage the community and maintain a usable product. Even companies with more resources can benefit from community-based bug fixes and optimization. Due to licensing restrictions, companies can not sell applications developed like this as proprietary, but it can be used to drive up demand for the base product and complementary software, hardware and services.

The concept of donated complements itself is not new in software business. In games industry, companies have for a long time provided users with toolkits to help them develop modifications, also known as “mods”, to commercial games, thus increasing the perceived value of the game.

Debugging

When developing new software, there is a constant trade-off between development speed and software stability. If the software is required to be completely bug-free, the required development resources and time will increase in magnitudes. If bugs and minor problems are accepted, development is cheaper, faster and more efficient as there is more communication with the user, earlier in the process. The further a project has progressed, the higher the cost of change [11], hence earlier communication saves resources and time.

In many open source projects, users tend to accept a greater deal of insecurity than in commercial software, and are often willing and able debuggers for the software, enabling high-speed prototyping development (30, s. 23). Also exposing a bug to a large user-base is helpful in

[11] http://www.agilemodeling.com/essays/costOfChange.htm

finding and fixing bugs fast. A famous quote from the book "The Cathedral and the Bazaar", "given enough eyes, all bugs are shallow", bases on the notion that the Internet enables a larger community of developers than can be allocated to a project by even the largest of companies.

Shared ownership

As open source software is often developed by the initial company, open source community and third party companies together, the ownership of the product is often blurred, as every contributor owns the original copyright to his or her contribution. Contributions are submitted with the chosen open source license, but the diluted ownership means that the original developer alone cannot pull back the software and change it's licensing scheme.

Infighting

Open source communities are often considered by outsiders to be serene congregations, where experienced, professional developers discuss technological nuances in a civilized manner. Quite often it is not so, but instead infighting is common and can often be harsh and personal [12][13][14][15]. It is not rare for participants to resort to verbal abuse and even threats of violence. Especially women tend to fall victim to verbal abuse, due their reduced aggression compared to male developers[16].

Even though developers in an open source community might have shared interests with a commercial company and among themselves, history has shown that the community will very seldom be unanimous. Developers working for companies tend to have financial incentives in keeping their behavior professional, as it helps them to work with customers and colleagues and keep their position. When programming pro-bono, developers have lowered thresholds for flaming and bickering, as the project tends to be more of a personal hobby than a financial investment. For example Linus Torvalds, developer and director of the Linux project, is notoriously famous for his obscene language and attitude towards those he deems unworthy of developing Linux[17].

If left unchecked, growing tensions risk tearing a community apart and either throwing it off-track, or one side forming their own fork of the software project, creating an identical project competing for same developer pool. Infighting is not limited to software communities, as other

[12] https://gigaom.com/2014/08/27/chef-engineer-leaves-the-company-after-receiving-death-threats-from-its-open-source-community/
[13] https://plus.google.com/+LennartPoetteringTheOneAndOnly/posts/J2TZrTvu7vd
[14] https://subfictional.com/2012/10/08/death-threats-in-open-source-are-not-occurring-in-a-vacuum/
[15]http://www.theregister.co.uk/Print/2014/10/06/poettering_says_linux_kernel_community_is_hostil/
[16]http://smarterware.org/7550/designers-women-and-hostility-in-open-source
[17] http://www.internetnews.com/blog/skerner/is-it-time-to-restore-civility-to-linux-development.html

non-technical crowd sourcing projects also exhibit this tendency for infighting. Wikipedia has had to adapt rules barring contributors from altering each other's works back and forth (57, s. 3).

Intensity and frequency of infighting depends greatly on the attributes of the project. Smaller communities are not overflowing with interested developers and easily consider every single developer important. When a small group is developing unpopular software, they cannot afford to consider other developers as granted, lest they find themselves alone. Successful and large projects with large user bases however have less threat of losing developer base. Hence they are more vulnerable to infighting, as technical choices often have deep repercussions.

A recent example of an enormous source of infighting in Linux community is the introduction of a new startup manager called SystemD to replace older solution named SysVinit. Discussion on the change has been extremely vitriolic and petty, with many public figures receiving personal attacks and even death threats [18].

Risk of no community

Gaining a successful and useful community around one's open sourced product is not in anyway certain. Statistically, of the 46,356 projects hosted on the SourceForge.net, 95% had 5 or fewer registered contributors (24). This comparison however is not completely honest, as only a minimal number of open source projects are based on an existing, commercial product. However, exposing an expensive project as open source heavily limits the appropriation options, and should the attempt not bear fruit, will turn out as an extremely destructive endeavor, especially under heavy commercial competition.

3.2.1 Community structure

Open source projects cover almost all industries, with contributors from every imaginable background. Therefore every community is unique, with unique contributor profile, politics, rules and dynamics. However there are some common patterns in how successful communities are composed and organized.

Hybrid communities

Though the common metaphor of cathedrals and bazaars draws a picture of open source communities working in some completely different dimension and under different rules than any traditional organization, reality is more complex. Most successful open source projects are neither bazaars or cathedrals, but hybrids combining features from both, thus combining the benefits of modularity and voluntary production (bazaar) and conscious planning and integrality (cathedral) (37, s. 4).

[18] http://linux.slashdot.org/story/14/11/16/2142244/longtime-debian-developer-tollef-fog-heen-resigns-from-systemd-maintainer-team

Open source communities consist mostly of pro-bono developers (10, s. 2)(21, s. 9), with occasional paid professionals and companies taking part. Especially companies in the related ecosystem that provide products or services around an open source project often support full time developers working on the project itself (46, s. 310)(10, s. 4)(4, s. 6)(25, s. 45). Most obvious example being almost every major tech company contributing to the development of Linux. Communities often don't distinguish between a corporate and an individual contributor, and there is no sense of direct competition between companies and the community (32, s. 5).

Nationality and ethnicity

Due to the omnipresence of Internet and globality of software, open source communities tend to have contributors from all over the world. Still, the geographic distribution is heavily concentrated around US and Europe (4, s. 6), an overwhelming majority (study estimates 97.5%) being males of an average age of 30 (35, s. 9). This skewness towards young white males can be seen as a factor contributing in community infighting. However, the situation is expected to change over time since countries like India are investing heavily in the education of software engineers.

Core team

Communities can be enormous, with thousands of developers. Still, in the center of every successful project is a core team of maximum dozen people, who are responsible for most of the new feature development and organization (1, s. 2)(12, s. 4). The core team is prone to change as politics and situations change and interests rise and fall. However, a large mass of off-core developers provides a sufficient pool of candidates eager to step up. The size of the core team is often dictated by a universal limitation of inter-personal communication and work division.

> *In many successful open source projects, such as Apache, meritocratic formation of the core team can be seen as a central factor in success. Developers striving for membership must display persistence, responsibility and capability. They can freely choose what they work on, and are thus encouraged to work on something that is either badly needed or where they exhibit special interest or skill in. This bottom-up prioritization process contrasts to commercial companies, where prioritization is given top down, often based on social and political issues, rather than technical (46, s. 344).*

Work division

Open source communities are fundamentally different from traditional organizations.

- Projects are not dictated by any formal plan, schedule or list of deliverables
- Work is not assigned, but chosen by developers
- Large number of voluntary developers working in temporary teams

To facilitate as low contribution barriers as possible, open source projects tend to be loosely structured, with contributors free to pursue their interests (37, s. 2)(37, s. 1). As the majority of developers are often not very active or reliable, and are mostly willing to take on easy and interesting tasks, it falls on the core developers to do anything that needs to be done in order to further the project (4, s. 1230), as they have little or no power to subvert motivations of individual developers.

In many similar crowd-sourcing projects, such as Wikipedia, a concept of indirect “ownership” manifests, denoting personal responsibility over a fragment of the system. “Owner” of a fragment, such as a module, is considered responsible for it, accruing credibility should the development of the system succeed (16, s. 4). It should be noted, that even though developers are often driven by a high sense of honor to complete a task for sake of their reputation, there is little the leadership can do to enforce arbitrary deadlines.

Modularity

Existence of the small core team despite potential for some kind of a large collective can be considered as result of Brook's law (49), which states that when the number of participants increases, productivity increases linearly, but effort required for communications increases exponentially. Therefore open source communities too fall victim to economics of diminishing returns, limiting the number of simultaneous participants.

Still, open source projects and commercial companies have beaten this law and produced complex software with large organizations. The secret lies in modular architecture, which allows the community to divide into teams, responsible for certain modules, or pieces of code. Successful modularization by the core team is a key aspect of a successful open source project.

Long tail

Although core team is responsible for most of new feature development, the extended community still provides substantial development support. Top decile of contributors is responsible for about 70% of new code (38, s. 204), whereas 75% percent of contributors ever only contribute once. There is also a noticeable difference in quality of contributions; the core team tends to develop new features, where extended community mainly commits documentation and bug fixes (56).

A large community is most suited for bug scouting or fixing purposes, as the large amount of use cases, eyeballs and computer configurations accounts for a greater chance for at least someone to spot and reproduce a bug, allowing it to be reported or fixed outright. In addition to providing more stable software, this frees up core developer resources to be used on feature development and bigger picture. Architectural design and development does not scale at all, but bug fixing, optimization and similar tasks scale well.

Open source values

Open source developers tend to share a set of values regarding equality, meritocracy and economics, often not completely aligned with commercial interests. A company disregarding these values in its actions and co-operation with the community faces severe competitive disadvantages, if they do not respect the intrinsically motivated rules of co-operation and trust of the community (4, s. 16).

As an example, Oracle, after buying up SUN Microsystems and therefore acquiring the ownership of MySQL, failed to listen to the vivid MySQL community. This resulted in MySQL forking into Maria DB, a completely open source version of MySQL, but with no ownership by Oracle. Maria DB quickly gained developer support and new features, competing with and reducing the appropriation opportunities of MySQL.

Leadership

For communities consisting of mostly pro-bono developers, the issue is leadership can seem irrelevant, due to there being no way to financially motivate them. Still, leadership of an open source community is very much a central factor in maintaining and organizing a community. Effective leadership differs greatly from commercial organizations as economic issues and motivations are different, and therefore, important considerations such as loyalty, satisfaction, and continuity play out differently.

Most open source communities are effectively meritocracies, where privileges and recognition is mostly based on individual’s capacity as a skilled developer; often disregarding other aspects such as social skills (46, s. 317)(30, s. 13). There is evidence of this meritocratic system often working effectively, with individual's status reflecting their effort and skills (51, s. 996). Reputation is established through quality contributions on a consistent basis that can lead to recognition and leadership roles, and is the only basis of authority in the community.

Strong leadership is important for an open source project to provide trust and dependability. Potential developers must believe in the long-term vision and quality of the leadership (17, s. 292). For many developers, the main promise of open source is often not the access to the source code, but trust in human leadership (17).

The “Leader” of an open source project often has practically no formal authority, having no economic or legal power over the community (37, s. 10). Still, the leader has substantial real authority, being a respected, fair and experienced member of the community, with political backing of many other peers. Maintaining this trust requires the leader to keep his objectives aligned with the community, not polluted by ego-driven, commercial or political agendas (38, s. 222)(22, s. 5). This may require the leader to accept meritorious improvements, despite them working against the leader’s original plans. The leader is often responsible for official releases and distribution.

Monarchs

In many cases, such as Linux, Perl, Gnome and Wikipedia, there is an undisputed leader, a monarch (38, s. 221). While many aspects of the project are delegated to 'lieutenants', these projects are strongly characterized by the presence of the monarch, who has the final say on any dispute (18, s. 204). While some other projects, such as Apache foundation, have successfully managed the direction of the project via a central committee, the lack of a strong central character may contribute to infighting and forking. The splintering of BSD-Unix has been attributed in part to absence of a single credible leader. Monarchs often have a history of extreme competence, such as being the original developer and still by far most capable contributor. As such, a monarch is often somewhat exempt from the usual, ongoing meritocratic process, with more power to impose their visions.

Forking

If a community has been divided in any major subject, such as direction, design, development methodologies, ideology or what not, it is vulnerable to forking. I.e. a splinter group can found their own community with their own repository and development process (38, s. 203). With an open source code base, there are no legal restrictions on the use of code in this manner. For the big picture this may be detrimental, as two communities might be developing the same features in parallel instead of with coordination, thus reducing the overall development efficiency.

Still, often forking can not be considered 100% harmful for an open source project; incompetence of previous leadership and inefficiency of development may lead to resources being allocated in a more efficient manner, much like in idealized free market. Without forking, developers might have left the project completely. As both code bases share the same history, it is often feasible to share code such as patches between forked code bases.

> *Over time BSD-Unix has splintered into many variants, each developed by a separate team. Still all variants share a common history and therefore can share code between themselves. Other example is the Linux distribution hierarchy, where many distributions are based on an older parent, forming a metaphorical tree of forks[19]. Ubuntu was originally forked from Debian and has been able to incorporate improvements in Debian to itself. As Ubuntu has gained momentum, Ubuntu developers have also started distributing their improvements, such as bug fixes, back to Debian, "upstream".*

For some commercial corporations however, forked, competing communities might be more of a poison, as there is no single party to negotiate with regarding direction and development. Preventing forking can be managed in multiple ways (30, s. 21). In the case of Linux, power lies with a charismatic central character with an established structure. Any potential insurgent group is likely to lack competitive edge due to lack of similar characters and existing structures and

[19] http://www.cyberciti.biz/tips/wp-content/uploads/2007/06/44218-linuxdistrotimeline-7.2.png

with relatively minuscule moment. Apache on the other hand is governed by a democratic coalition, which itself can change to suit requirements.

Derivatives

The tendency of communities to fork can also be harnessed to a company's advantage. For example Canonical, a major end-user Linux distributor, has supported all community efforts to fork its main distribution, Ubuntu. Instead of community forking, alternative versions[20], “flavors” have been developed as derivatives, and are even officially supported by Canonical. Ubuntu has been forked into a dozen or so versions, geared toward education, low-end machines, music production, or simply sporting different graphic user interface. Canonical maintains good relationships to these communities.

Social pressure

Even when the community has no economic or legal power over it's contributors, vested interests often guarantee that contributors consider their credibility and reputation important and are therefore susceptible to social pressure. Sanctions such as flaming, spamming and shunning can be used to force members change their behavior or leave the community. The effect is not unlike to that in non-professional team sports, where the success of the team depends on its members, despite not one of them having financial motives.

Communication

Due to geographical distribution, developers rarely, if ever, meet face to face. Communication through audio is also infeasible due time zone differences. To facilitate asynchronous, decentralized communication, open source projects almost always fall back to traditional messaging systems such as mailing lists, IRC, wikis and forums. These tools are ancient in scope of software as a whole, but are still extremely popular in distributed projects.

Lurking

Often a significant period of passive observation in forums and mailing lists, from weeks to months, is required before an interested individual eventually contributes to project’s technical discussion or submits code (59, s. 1227). One study discovered that on average, 23 emails were needed before an interested member actually became a contributor. This “lurking” period can be considered as the phase when a person learns about the project, and weighs pros and cons of contributing.

3.2.2 Developer motivations

Since the value of open source development mostly comes from a dynamic community and there is no silver bullet to motivate and kick up a community, the issue of understanding developers' motivations turns out critical. Especially so, since motivations of developers in open source projects differ greatly from motivations in commercial context (51, s. 984). Successfully

[20] http://www.ubuntu.com/about/about-ubuntu/flavours

internalizing developer motivations allows a company to better market the project to raise it's perceived attractiveness.

In general, developer motivations can be divided in two types, extrinsic and intrinsic, and three categories, economic, social and technological (4, s. 5). This division to three categories is intended to be more of a helpful tool, and not a logically sound model. Motivations are commonly aligned and indirectly reinforcing. For example, learning to be a better programmer improves the contributor’s career (extrinsic motivation), but also provides the joy of success (intrinsic motivation).

Relative significance of various motivators is not clear, and many studies have shown varied results; some finding intrinsic motivations like enjoyment most significant, while other have discovered economic issues, such as career signaling, to have the most weight (37, s. 11). It can be assumed that motivations vary highly on individual level, as well as between communities.

Examples of motivational factors

Economic motivators include considerations of both money and time. In short term, developers may receive monetary rewards from sources such as bountysource.com or individual companies for adding features and fixing bugs. In the long run, open source development is an investment to skills and portfolio, and hence to future career options. Open source projects are extremely easy to access, requiring little to no investments in time or resources, which lowers the opportunity costs.

Social and psychological motivators revolve around the basic human needs, for which open source is just a channel. Many developers enjoy software development itself, while some may feel delight simply from altruistic drives by helping others. A community also provides a sense of belonging and enables gaining status and reputation among peers. Besides, ideological drivers may encourage developers to share their time and skills towards a perceived ethical goal.

Technological motivators are about intrinsic self-development and understanding, which are common motivators for scientists and engineers in general. Open source projects provide a good framework for learning new technologies, programming languages and development methodologies. Furthermore they offer an efficient, though sometimes harsh, feedback channel in form of the community. Often developers simply improve an application they use themselves, practically scratching a personal itch.

Monetary rewards

While most open source developers work for no monetary gain, there are channels one can use to monetize their skills. Commercial companies that use open source have financial incentives to get bugs fixed, and should they not have competence of their own, they can enlist their requests in open source forums or sites[21][22], which facilitate monetary transfers for open source contributions. Evidence points out that in open source projects extrinsic motivations, such as financial compensation, improve contributor performance, but do not reduce the value of intrinsic motivations, such as joy, but instead reinforcing them (51, s. 994).

Low opportunity costs

For a developer, investing effort into a project can be risky, if there is a risk of the owner afterwards closing the code and selling improvements without reciprocity. Open source licenses negate this risk and guarantee the developer access to all open sourced modifications.

Another issue for a casual developer are entry barriers, should a project require some sort of certification, special tools or the general contribution process is heavy. Most open source projects attempt to minimize entry barriers to facilitate the inflow of new developers.

Future career benefit

As many companies utilize open source in their business, there is a steady demand for developers knowledgeable in these applications. Success, reputation and quality code in an open source project is a strong signal of a developer's initiative and relevant skills. This signal is stronger the more visible the performance is and the more informative the performance about individual's talent is (38, s. 214). Higher ranking in an open source community can lead to wages 14 to 19 percent higher, even if the job is not directly related to the open source project in question (37, s. 10). Faith in signaling talent through open source development is stronger if related commercial companies have actual interest in the open source project and perceived capabilities to inspect the code (4, s. 3).

Enjoyment

Unlike in many other professional fields, software developers often consider programming fun enough to spend their free time on. Open source projects offer developers a platform for finding challenges and testing their mettle. Sense of joy, creativity, autonomy, competence and flow has the most impact on a developer's commitment and performance (35, s. 16)(51, s. 986).

Altruism

Evidence points out that developers contribute out of sense of altruism (30, s. 23), ignoring relevant economic calculations in their decisions. Acts of altruism can be considered contributing to internal values such as self-esteem.

Sense of community

Belonging to a community is on of the most basic of human needs. Open source projects form communities where individuals may interact with similarly minded peers in a meaningful environment, jointly aiming for a common goal. Communities are self-reinforcing, as individuals

[21] https://freedomsponsors.org/
[22] https://www.bountysource.com/

search for similarly minded communities, but also internalize values and function of the community, growing as a part of it (42). When open sourcing the source code of Linux and therefore forfeiting all options for profiting on it, Linus Torvalds stated that by open sourcing the code, he could build a community he wanted to be a part of (4, s. 9).

Ideology

Many developers conform to the values of open source, thinking that software should not be proprietary (19) and feel proud for being a part of the open source movement (23). Stallman in his GNU Manifesto originally went as far as to consider the use of proprietary software dishonorable (4, s. 10). While extreme antipathies towards commercial companies are diminishing over time, tensions still persist due companies often having opposite or unrelated interest compared to individual contributors.

Status and reputation among peers

Identification and status with the community can be considered direct components in a developer's utility function (30, s. 24)(3). As open source communities tend to be meritocracies, where individual's status mostly depends on actual performance, status and reputation inside the community can both provide a person with the sensation and an actual meter of excellence. Person's development performance and community status form a mutually reinforcing feedback loop (51, s. 997).

Though developer motivation can be reinforced by publicly praising individuals for good performance (38, s. 218), community status can be considered a self-sufficient motivation, even without any external, e.g. career, effects (16, s. 1). Open source communities also always guarantee that a developer's contributions will stay visible.

Also as communities have zero de-jure power over members, importance of public recognition and credit gives effect to the community's rules, senior members' opinions and sanctioned flaming (4, s. 18).

Learning

Open source development allows developers to freely read and inquire about each other's code, as well share and receive as meaningful, constructive feedback. This translates to a suitable environment for development of personal skills and human capital (4, s. 26).

Working with a bleeding-edge technology

Professional developers are often employed in commercial projects, where technology choices have been made years, or decades, before. This locks projects in with old technology and rigid architecture. Still, software world evolves over time and new technologies appear. To keep up with the progress, developers often have side projects where they dabble in and learn upcoming technologies. Open source projects, provided they are based on a new technology, offer a framework where developers can learn in a meaningful, realistic and practical environment.

Fads

Similarly to the scientific community, open source communities, and software world as a whole, are susceptible to “fads”. Often some specific new technology, development methodology or field comes out as fashionable, and projects related to it have sudden spike in perceived value for contributors, as experience in the current fad can have a strong signaling effect (38, s. 214).

Scratching a personal itch

Solving personal problems is the oldest of motivations for developing open source; in the beginning of the software era, there were few tools and programming languages and none of them could be considered complete. Developing any software required one to make and improve one’s personal tools. To some extent this motivation still persists, as developers often need small features or bug fixes in their tools and frameworks.

Similarity to scientific community

Motivations in open source community can be compared to the scientific community, as production of software, like scientific discovery, is often a form of intellectual gratification with intrinsic utility and little to no financial remuneration (3, s. 1245). Scientific communities also rely heavily on the concept of credit and reputation, where community reputation can translate into better social position, grants, data and publications (16, s. 2).

New project appeal

Fresh projects often find it easier to attract contributors. This is due several reasons (38, s. 221).

- New projects often incorporate newest technologies, with higher learning and fad value.
- In an incomplete project, there are still significant architectural and technical challenges, whereas more mature project may have fewer.
- Should the project gain momentum, early contributors have a higher chance to gain recognition, visibility and status for their early contributions.

Signaling visibility

Regarding signaling performance, open source projects have three distinct features that make them more appealing for developers valuing signaling (37, s. 9).

- Outsiders can see the contribution of each individual and whether that component
    - Actually works.
    - Was hard to accomplish.
    - Was addressed in a clever way.
    - Whether the code can be reused later.
- Open source programmers bear full responsibility for their sub-project, with little to no interference from superiors, and managing such a project with success is a sign of one’s ability to follow through with a task.
- Many elements of source code are common across open source projects and accumulated knowledge can be transferred to new environments.

3.2.3 Nurturing a community

Parasitic approach

Approaches to community interaction are aplenty and lie on a continuum, with two extremities: The company only focuses on its own benefits, not taking into account the dynamics of the community. While leeching is an efficient way to drive down a community and irritate developers, it may be a necessity for a company attempting to shave down costs. Leeching from a strong and healthy community is less likely to cause badwill, while still saving the company's resources.

Symbiotic approach

Opposed to the parasitic approach, a symbiotic company tries to develop both itself and the community. More weight is put on the community in the business decisions, but also more on the company by the community. It is required for company to be involved in community development, as legitimacy to influence the community cannot be gained from having a formal role in a firm, but on the status gained in the community, based on its norms and values.

Minimize entry barriers

When a person is lurking on an open source project, he is susceptible to practical hurdles and friction; if the project and the discussion are difficult to grasp, the documentation is lacking or mailing lists are poorly organized, joining development might get postponed or skipped altogether (59, s. 1231). Combating this "contribution barrier" is essential for facilitating an easy access into the project and keeping the perceived cost of contributing as close to zero as possible (4, s. 6). Facilitating easy contribution and access to the source code is also essential for building the ecosystem, as it encourages other commercial companies to take part in the ecosystem around the project (25, s. 45).

Commit-then-review

One common contribution barrier is the common review-then-commit-procedure used in many projects by default. The idea is that when a contributor has developed a patch or a feature, it is first placed into public review, and after passing it, committed to the code base. This method is called review-then-commit, RTC.

RTC methodology provides the community with better control over the code base, as every insert is studied beforehand. However, this control comes with a heavy cost; reviewing all code before a commit is very time consuming (51, s 542). This adds to an enormous overhead when committing short, one-line bug fixes to software still heavily under development. Demise of Nupedia, original form of Wikipedia, was credited to a rigorous, forced peer-review process (16, s. 5).

An alternative method, commit-then-review, CTR, can be used when an already trusted developer feels confident in what he is committing. This method is more susceptible to bugs, but reduced overhead often makes up for this, as bug fixes are also easier to commit afterwards.

Provide vision and leadership

Providing leadership and vision is crucial for an open source project (18, s. 203). Developers never wish to invest time in a project they don't believe to have a future. Knowledge that a commercial company is fully engaged with a project is a strong sign of momentum. Even if the company's vision might slightly deviate from that of individual’s, the project is still destined to carry on, thus maintaining value for performance signaling. Long-term goals help developers to maintain faith in the project, despite short-term challenges.

Give credit

Gaining credit for work done is a central motivation for many developers (51, s. 997)(37, s. 218). When designing an on-line community, a meaningful structure is required to display and reward contributors for their performance. This can be achieved by rewarding performing members with status symbols such as forum icons or ranks (many forum engines display the member's post count), sharing responsibilities, devoting website space for distinguished contributors or facilitating references for potential employers etc. (16, s. 5)(51, s. 997).

Apache foundation maintains public pages, which give recognition to core members for their achievements[23]*. Apache also maintains lists of members who have contributed by, for example, identifying a problem without proposing a solution.*

Intermediaries

Fears of commercial take-over of a project can be reduced by utilizing an intermediary, trusted third party companies[24] or foundations[25] that take responsibility for the project and thus provide contributors with a guarantee of the project's future and values. Establishing a non-profit organization for managing the project can be used to alleviate conflicted interests and provide more autonomy to the community.

Gnome foundation is a non-profit organization that takes care of the open source Gnome (desktop environment) project. It provides the community with a democratic process with elected representatives, guarantees transparency on future decisions, handles communications with media and corporations, and acts as the legal entity that can accept donations and invest money on behalf of the Gnome project (18, s. 205).

[23] http://httpd.apache.org/contributors/
[24] http://www.collab.net/
[25] http://www.apache.org/

Collab.net is a commercial company that "certifies" commercial open source development programs and helps companies to contact and select open source developers and settle disputes (38, s. 226).

Set up code distribution channels

Even though the requirements of open source licenses' are satisfied by mailing a CD with the source code to anyone requesting it, this is hardly optimal. A well-planned code distribution strategy is important for minimizing all access and contribution barriers (22, s. 5). The code should be available at least on project's web site, wrapped in a common format, such as zip or tar, preferably organized in versions and sub versions with change logs attached. In addition, distributed repository services[26] offer a good platform for many projects.

Over time, Github has attained a position as a default repository, due to its practicality, good user interface and the quality of Git, an open source version control software itself. It is a distributed repository system that allows members to directly clone the main repository with minimal hassle, and offers an easy way to pull changes back to the main repository. Note that Git, the repository system Github is based on, is only one of many distributed repository systems in use, and Github is simply a commercial company built around it.

Maintain Todo-lists

To begin development, a potential contributor requires some guidance on where he could begin. Open bugs are an easy way for a newcomer to get acquainted with the code base and the architecture of the project (18, s. 206). Therefore, maintaining a clear, easy-to-access bug database is necessary for facilitating an easy entry. Every module should also have a separate, related Todo, listing of required activities, features, bugs, testing, documentation etc.

Modularize architecture

Modularization of a program means that the program is divided into logically separate pieces, modules, which have their own functionalities and little inter-dependency. This allows multiple teams to work on small parts of the software without unintentionally interacting with each other's code and without requiring a specialized developer to completely comprehend the whole entirety of the program (22, s. 5).

To fully utilize the potential for a large developer base, the company needs to split its code into modules (18, s. 213). Success in this architectural effort is of utmost importance to a project's ability to attract developers and efficiently produce good quality software (38, s. 220).

[26] https://github.com/

Code release and preparation

The question of whether to release an internally developed product as open source, or develop it as such from beginning, is hardly ever asked; other, historical, economic and practical issues have often forced the decision long before.

To gain momentum on the project, the company must provide a critical mass of code, which the community can get excited about and build upon. Enough work must be done to prove the feasibility of the project and original developers' skills (38, s. 220). Furthermore a relatively finished project may find it hard to attract developers, if there are no intriguing programming challenges left.

Often an internally developed, or evolved, code base can contain harmful or embarrassing skeletons that need to be cleansed. Comments and token naming should not be obscene and obvious emergency-patches and other anomalies should be removed (25, s. 47). Poorly commented code with poor technical quality gives a detrimental message of the project's merits. Still, exposing the code to the community adds significantly more eyes and quickly exposes bugs and other issues, which can then be fixed.

Legal restrictions

Should the software contain technology licensed from commercial corporations, or contain already open sourced code, care is needed in choosing the proper license. Third party commercial code can legally block open sourcing completely, and viral open source licenses may require the whole application to be open sourced (25, s. 50). Some countries have country-specific restrictions on exporting software. For example, the United States has restrictions on exporting cryptographic material (31).

Document the code

The bigger and more specialized a project is, the more crucial it is to provide proper documentation and code commenting. In extreme cases companies have open sourced code bases of over half a million lines. Without proper guidance, scaling such projects is absolutely unfeasible (37, s. 12).

Adapt development process

The method of open source development is significantly different from internal development process in a commercial company (18, s. 213)(25, s. 50). It is ill advised to expect the same process to work with a community not bound to schedules and financial motivations. Adaptation needed depends on many variables such as old, existing development process (agile vs. waterfall), nature of the product, customer requirements and schedule, size and structure of the community etc. In general, a company should be prepared to bear the ultimate responsibility of the project and take care of urgent issues such as critical bug fixes, whereas less urgent development can be more easily trusted to the community (25, s. 47).

Fringe benefits

Fringe benefits, such as developer summits, free licenses, early access programs, equipment and small gifts can be used to encourage certain types of behavior, such as bug hunting. However, creating sustained interest with fringe benefits alone is unfeasible (10).

Adapt organization

Commercial companies often have strong hierarchies, where decision processes are long, with ultimate power over architecture and design residing in higher echelons of the company. This is a stark contrast to open source communities, which are often very flat and discussion is not as limited to a traditional chain of command.

When dealing with an open source community, companies need to adapt a flatter organization in order to draw the developers to the core of the project, both to gain more insight and ideas, and to offer the developers a feeling of importance and value (18, s. 216).

Preventing forking

Maintaining the unity of the open source community is both a political and technical issue. Leaders need to discourage, or at least track all independent strains of the code base carefully (22, s. 5). Sometimes motivations for forking can be political, with some members of the community disliking the leadership or management processes. In these cases, the solution is most likely to be found with mature political means, such as discussion and democracy, or less mature, such as flaming and blaming. In practically no business scenarios can the latter ones be considered a smart move. In open source communities however, they may sometimes prove useful.

Forking an existing code base may also be a technical necessity. One specific subgroup of the community might require some features that are incompatible with the majority, thus encouraging forking. In these situations, the solution should be looked for in the technology; perhaps the software can be modularized in a better way to suit both requirements, and if not, maybe the fork can be managed better as a derivative.

> *A recent example[27][28] in Linux communities is the possible change from venerable SysVinit system management daemon to a newer, modern System. News SystemD is not a simple upgrade, and has a host of drawbacks, as does the older SysVinit. The CEO of canonical, Mark Shuttleworth, went to great lengths[29] to make a democratic, transparent decision regarding Ubuntu. Another distribution, Debian, ended up getting forked over the issue[30].*

[27] http://wiki.gentoo.org/wiki/Talk:Comparison_of_init_systems
[28] https://fedoraproject.org/wiki/SysVinit_to_Systemd_Cheatsheet
[29] http://www.zdnet.com/after-linux-civil-war-ubuntu-to-adopt-systemd-7000026373/
[30] https://devuan.org/

Devote personnel

In order to help development and help subtly control the community, companies can devote employees to work on the project (51, s. 987). Even when employed by a commercial company, paid developers are typically considered as peers in the community and judged by their merits. Still, as full time developers, paid developers have a good opportunity to accumulate a great deal of experience and insight in the project, gaining respect with their performance, allowing more control on the project and also benefiting the company's reputation (10, s. 10).

Maintaining reputation

The psychological contract between contributors and companies is based on shared beliefs, reciprocity and trust (4, s. 17). Maintaining the contract requires the company to commit to the project and to respect the rules of the community. Failing to maintain a good reputation backfires as competitive disadvantages, and as lost co-operation and control (2, s. 6).

Reputation is an important enabler of control for a community (10, s. 10). Without goodwill the community is less likely to accept or value the company's goals and leadership, weakening unity of the community and endangering it to forking.

Set up communication channels

Open source communities communicate through the Internet, via mailing lists, IRC and forums. Providing and organizing this infrastructure is essential for minimizing friction and entry barriers (10, s. 10).

A sense of community is mainly built by continuous communication between members. IRC and other equivalent chat room applications offer an effective platform for casual “water-cooler” conversations, instilling trust and bonds. Mailing lists on the other hand tend to work as de facto archives where public discussions are lead and simultaneously archived for future usage and for new members to learn from (18, s. 213).

Many open source projects tend to arrange co-located meetings, “developer summits” periodically. While expensive to arrange, these summits are efficient in solidifying the community and allowing improved, casual communication and community building.

Provide tools

Besides communication infrastructure, an open source community can benefit greatly from other related tools (33, s. 1). Developing specialized tools that ease development, and especially testing, can be a feasible mission for the company, while improving the efficiency of both internal and community developers. Investing in improved tools, instead of the code itself, has scaling returns with the size of the community.

Conflict resolution

Conflicts are an endemic feature in the open source community (60, s. 3), since developers and related companies often have mutually misaligned incentives. If tensions are not dealt with, they can evict less confrontational members and leave the community split in factions, in the worst case leading to competing forks of the project. Providing a neutral ground for discussions or alternately providing the community with a benevolent dictator can help members vent their frustration, while still ensuring unity of the community.

Place bounties

Though most developers are motivated intrinsically, sharing monetary rewards for development tasks is still a viable way of influencing the community (10, s. 10).

3.3 Domain: Technology

3.3.1 Quality of open source software

Bugs are unavoidable

In software development, bugs and vulnerabilities are a fact of life. Producing absolutely side effect free code is magnitudes more time and resource consuming than producing software with acceptable levels of issues. For long-term perception of software quality, the most important factor is not the initial bug count, but instead how quickly apparent bugs are fixed. A rapid release rate of open source software allows fixes to be deployed quickly, potentially an order of magnitude faster than those of commercial software (28, s. 13)(46, s. 317)(12, s. 8)(37, s. 19).

Motivation for quality

In proprietary software companies, code is rarely read by anyone except the developer himself, and writing exceptional quality code is more arduous than writing things the easy and fast way, as commercial management is often more interested in schedules than code quality. As open source enables peers to pry on ones code, good quality code has high signaling value and motivates developers to keep the quality high (1, s. 8)(51, s. 541).

Many eyes

As mentioned earlier, open source community offers a good foundation on product quality control (46, s. 341). Firstly, due to often having a large installed base, discovering bugs is likely to happen sooner than later. Secondly, due to a commonly technical user base, it is probable for a user to be able to reproduce the problem and create a useful bug report for developers to analyze. Useful bug reports trivialize the effort required to fix bugs.

High technical quality

Open source development has been proven to be able to produce high quality code efficiently, using less lines of code than commercial competitors (28, s. 13)(46, s. 331). As a result, open source software is often more robust, maintainable and easier to develop further. This is not to say, that all open source code is of high quality. Instead it should be thought that open source methodology is capable of producing high quality software.

Weak graphical user interface (GUI)

Direct result of a highly technical developer and user base is that open source projects typically have difficulties producing competitive graphical interfaces, as technical capability and artistic skill tend to be mutually exclusive to some extent (28, s. 13). Many projects are designed to be used as “headless” services via command line, with zero graphical user interface, and the GUI is added as an after thought, if at all.

Many successful, commercial Linux distributors add significant value to their distribution by utilizing professional interface designers and investing their resources in improving graphical interfaces of open source software. In fact, open source development can be considered as a complementary to commercial development, as open source provides raw material, which commercial actors can further refine into a product (62).

Effect of commit-then-review

As CTR procedure allows developers to commit relatively untested features with little overhead, the code base is more vulnerable to bugs. In this sense, RTC-CTR is about a trade-off in overhead versus reliability. However, in a well functioning project with healthy management and developer motivations, the CTR procedure does not produce significantly more bugs (51, s. 544).

3.3.2 Development dynamics

Productivity

Evidence shows that open source development processes have emerged as an efficient solution to reducing development time and reducing design, implementation, and quality assurance costs for certain types of software (46, s. 337). Especially infrastructural software, e.g. operating systems, compilers, middleware and editors have benefited from open source methodology (65). The efficiency of the open source model is also implied by efforts of commercial companies to emulate it internally (30, s. 29)(51, s. 9).

Microsoft encourages exchange of ideas by building software teams and cultivating developer networks within the company, in a sense creating an internal peer community to appraise and debug each others' software. This is a remarkable deviation from typical silo-style process, where communication between peers goes through their managers.

Hierarchical organizations also often tend to suppress critique, inhibiting people from challenging each other's ideas due to reputational and financial repercussions. Open source development avoids most organizational pathologies hindering efficient software development in commercial companies (37, s. 19).

Scalability

Brooks' Law of scalability states that adding manpower to a late software project delays it even further. The problem arises when the amount work getting done scales linearly with increasing developer base, but the project complexity and need for communication grow geometrically (65). Conservative company politics and accountabilities often forbid or hamper effective division of labor and responsibilities between divisions and individuals. Especially the delegation of authority and responsibility is almost nonexistent. As a result, software industry is filled with stories where 10 good developers have blown away competitors with thousands of mediocre developers.

Early history between Microsoft and IBM is an example of this, IBM embracing its "masses of asses"-approach (term coined by Bill Gates), while Microsoft had a small team of quality developers. Still, Microsoft managed to oust IBM from the market for desktop operating systems.

Despite consisting over 2 million lines of code, Linux has been growing at a "super-linear" rate for several years, bypassing the common phenomenon where development slows down as the software grows larger (12, s. 1). Moreover other open source projects, such as the Apache project have overcome Brooks' Law and achieved scalable efficiency of development.

In contrast to development, software debugging and quality assurance do scale up as the number of developers increases, as testing and debugging does not require precise coordination, nor does it introduce collisions if two individuals test overlapping functionalities (65).

Open source methodology bypasses Brooks Law with two central features. First, projects are often organized into core and periphery teams, where a small core is responsible for non scalable operations, such as design and planning, leveraging improved communications provided by a smaller team size. The periphery team on the other hand helps with development, debugging, documentation, testing etc. Secondly, large open source projects are often divided into distinct modules, interaction of which is well defined and understood. This allows multiple small teams to operate on the same software without requiring extensive cross-team communication.

Scheduling

Commercial companies are often required to enforce schedules for development processes due to customer requirements and promises. Paying customers expect companies to provide and adhere to schedules, as their own business often depends on it.

In open source development however, lack of commercial pressure and developers mostly working on their off-hours make projects immune to time-to-market pressures. As a result, a system need not be released until participants are satisfied with system maturity and stability (33, s. 2)(30, s. 14). If a company provides their customer with support to an open source project, they may be required to develop the product on their own to match customer schedules.

Open source processes can still be adapted to fit commercial requirements. Mozilla's hybrid internal-community development introduces more commercial-like aspects to development, including more thorough code review and submission process. Mozilla uses a 30-day release cycle that is more akin to commercial methodology, than open source development (46, s. 342).

Prototype code

Commercial development processes often adhere to strict testing standards, such as unit tests for every functionality and high code coverage. In part, this is to secure project continuity, in case the developers change or the product is sold. Automated test cases allow new developers to make changes to a system without fear of breaking something somewhere else. Writing tests however, is time-consuming and arduous, and often not favored by open source developers.

> *To distinguish prototype versions, Linux development has adapted a system[31], where even numbered releases, such as 2.0 and 2.2 are considered major releases. Odd-numbered, such as 2.1 and 2.3 on the other hand are developmental versions with new, unrefined features. This practice allows the development process to cater for two very different audiences: developers and enterprise customers (32). These two branches are effectively developed simultaneously as the development and stable -branches. Stable releases contain updates and bug fixes based on previous stable releases, whereas development branch includes features, which once mature enough, are migrated stable branch (33, s. 2).*

Planning

Author Steward Brand coined the phrase “All buildings are predictions. And all predictions are wrong” (60, s. 16). Originally this considered actual buildings, but it can be adapted to fit processes of building software; “All applications are predictions. And all predictions are wrong.” Even simple software is often far too complex to be fully designed before development. As software is developed, new details, problems and opportunities are recognized. “The devil lies in the details”, so to speak.

Historically, software development processes have embraced the idea of comprehensive planning, incorporating distinct consecutive phases: requirements specifications, architecture design, development, testing and maintenance. This ideology[32] is responsible for many major failures of software history, as it relies on the idea that some group of people is capable of divining all possible use-cases and requirements without any prototype system whatsoever. Typically this planning group might not even contain representatives from the customer or user segments**.** In addition, the software environment is in perpetual flux as new components, technologies and requirements come and go.

Most successful development methodologies are based on the idea of agility[33][34] that distinct tasks, such as requirements engineering, design, development and testing are done simultaneously, often in small, iterative “sprints” of several weeks.

[31] https://lkml.org/lkml/2005/3/2/247
[32] http://en.wikipedia.org/wiki/Waterfall_model
[33] http://agilemanifesto.org/
[34] http://en.wikipedia.org/wiki/Agile_software_development

Open source development often represents the most extreme of agile development, where developers are quick to react to requirements and feedback extremely quick. On the downside, planned evolution, testing and preventative maintenance may suffer as open source development does not encourage careful reflection and refactoring code, and code quality is often maintained only by massively parallel debugging (i.e. many developers using each others' code), rather than by systematic testing (33, s. 2).

Projects often evolve stepwise, instead of basing on regular or even initial requirement engineering phases (18, s. 208). Still, open source development offers some benefits to the planning process. Once a problem is identified and community is looking for a solution, the problem is often not about finding the solution, but instead, selecting a solution from various options, each with their own drawbacks and benefits (46, s. 319). Open source encourages wide discussion to reach a consensus, scouring different opinions and aspects among multiple developers to provide a generally “best” solution.

Feedback systems

Success of software development is often not about preliminary planning, but reactivity. Well organized open source projects, such as Linux and ACE + TAO, have leveraged the short feedback loop between core and periphery, even bringing response times to bugs down to a matter of minutes or hours (65). Propagation of these changes is supported by use of modern version control tools, such as GIT[35], which allow users in the periphery to easily synchronize updates in real time.

Feedback mechanisms in the open source process differ remarkably from those typically present in commercial companies. Companies such as Microsoft often have customer representatives and feedback channels that any customer can utilize. Channels like this are most used by noisy, ignorant customers, who, as paying customers, must be served accordingly. This results in dumbing down of the software and inclusion of unnecessary bells and whistles. The Linux community on the other hand receives feedback via mailing lists and forums, which are mostly utilized by highly technical individuals, emphasizing messages of “smartest” customers. This comes with the price of correspondingly filtering out smart feedback from non-technical users, who have difficulties getting heard in an overly technical environment (14).

In contemporary time-to-market driven economy, few software companies can afford long quality assurance cycles. As a result, almost every computer user is effectively a beta-tester of software that was released before it was extensively tested and debugged. In the traditional, proprietary model, premature releases yield discontent users who have few options when problems arise with the software they purchased. As their only option is often to devise workarounds, they have little incentive or options to improve development of given products. On the other hand, open source development empowers users. Short feedback loops encourage users

[35] http://git-scm.com/

to help, as they are rewarded with quick fixes to identified bugs. Users also have more options, as they have direct access to the code and can locate bugs, craft patches or provide test cases that allow easy isolation of bug (65).

> *Steam's Early Access program*[36] *acts as an example, in that users are willing to use severely unfinished products, given that they are informed about the limitations. Early Access allows players to access games still deep in development.*

Management and accountability

In most commercial software companies, development is conducted by managers who are also responsible for the design. This is often a hierarchical requirement, as many tall organizations require a single point of accountability for a given organizational unit. This introduces multiple issues, such as that of scalability, as a manager can only supervise so many programmers (60, s. 16). To facilitate more scalable development, it is necessary to push responsibility and accountability deeper, and to let developers run on their own to some extent. Scalability is impossible in a tightly supervised environment, and tight supervision is inevitable, if the supervisor is held fully accountable for all choices and actions of his subordinates.

Core teams

Developers of an integrated application are required to cooperate tightly together with a relatively detailed knowledge about each other's actions. Without sufficient coordination, the team is vulnerable to mutually incompatible design choices and changes in the code. Since the core team needs to act as a single entity, it is easily overwhelmed by a coordination and communication overhead, which typically limits the size of an effective team to 10-15 developers (46, s. 328)(30, s. 28). This is not a strict limitation for the amount of people capable of writing application code, but restricts the number of people that can make mutually dependent design decisions.

One notable attribute is that despite working in a tightly knit group, core teams are often geographically highly distributed and rarely meet each other in person. Instead, communication happens over Internet, and practically all exchange is recorded in electronic form (46, s. 313).

Reusing existing libraries

Open source licenses tend to be mutually compatible, meaning that developers of open source products can incorporate existing open source code in their product. As many company developers utilize and work on the most common applications, there is a growing ecosystem of quality software. Knowledge and efficient application of this ecosystem enables developers to avoid reinventing the wheel and significantly improve the speed of development and software quality (32, s. 7).

[36] http://store.steampowered.com/genre/Early%20Access/

Many programming languages have large, centralized repositories of existing components, which can be included in an application with minimal labor. Examples include languages and repository systems such as Ruby (gems[37]), Perl (cpan[38]) and Java (maven[39]).

3.3.3 Modularity

"The key is to keep people from stepping on each other's toes"- Linus Torvalds

In software development, modularity refers to a feature[40] of the software architecture, namely how the application logic and functionality is divided into separate, logical sub-systems. Modularity greatly influences multiple aspect of development, such as how easy the application is to understand, maintain, develop further and how many concurrent developers may work on the system.

Specialization

Efficient modularization enables developers to concentrate only on some parts of the system, allowing high specialization (37, s. 11). Without effective subdivision of tasks, every developer is required to fully know the entire system to avoid regression. In even moderately large software projects, this is practically an impossible task, and high specialization is paramount for efficient development (59, s. 1230).

Specialization benefits occur even if a project only contains a single head developer. As the project matures, focus of development shifts on new segments. Again, without efficient modularization of the application, the developer is required to consider the whole complex system all at once. With modularization, he is allowed to forget implementation details of other modules, and just consider the inter-module interfaces and the current component at hand

Linus Torvalds has testified to the value of modularity, and wanted Linux kernel to become as modular as possible. The open source development model practically requires this to enable parallel development and preventing clashes. Besides open source developers don't work in shared premises, so often responsibilities over subsystems are clearly divided. Without modularity to prevent collisions, the developer is required to check all other changes made in the system in case there are incompatible changes by other developers. Modularization enables maintainers such as Linus to also accept new modules they don't yet fully trust, because they can trust the fact that the module cannot alter the system outside its own borders.

[37] https://rubygems.org/
[38] http://www.cpan.org/
[39] http://search.maven.org/
[40] http://en.wikipedia.org/wiki/Modular_programming

Entry barriers

As a project matures, it grows more complex, and in the end, only a few people who have been actively involved in its growth have any chance of fully understanding the architecture and effectively contributing code. In the case of an open source project that relies on an influx of new developers, a requirement to understand the whole application is effectively a death sentence, as no new developer can cross the entry barrier. On the other hand, if the project incorporates a modular architecture, new developers can choose a tiny part where to start (59, s. 1218). In Mozilla's case, redesign of software to a more modular architecture was followed by an influx of new community developers (41, s. 28).

Managing complexity and concurrent engineering

From a leadership standpoint, managing a large and highly complex system provides a variety of its own problems. Ages ago, Sun Tzu wrote about the issues of commanding large armies. However, he found the solution almost trivial, stating that ”The control of a large force is the same principle as the control of a few men: it is merely a question of dividing up their numbers”. Meaningful division of workforce along module borders allows leaders to better comprehend and organize the responsibilities and interactions of workforce and also increasing project's transparency (59). The success of open source projects depends on maintainers' ability to divide the project into sub-projects, which require minimal inter-coordination and thus reduce organizational complexity (18, s. 203)(49, s. 6). Decomposable technology enables decomposable development organization (37, s. 14).

Apache's approach to coordination seems to work extremely well in small projects: the Apache server itself is kept minimal, and all functionality beyond the most basic core is added through independent modules that are developed in open source projects of their own by separate developers and separate leadership (46, s. 342). This model relies on clear modular architecture that allows interaction via interfaces only. There are over 100 modules distributed by third parties, many of which contain more lines of code than the core server.

As Linux gained momentum, Torvalds became overwhelmed by the amount of code changes, and, as project members noticed, "Linus does not scale". To deal with this, Torvalds delegated large components to his "Lieutenants", who further delegated to "Area" owners. (41, s. 24)

One particular feature of GNOME is that it is being created by hundreds of contributors, making it one of the largest open source projects. This is also enabled by the effective division of work.

Adaptability

Basing technology on a highly modular software also brings benefits in the form of adaptability and customizability; if a certain module is suboptimal or lacks functionalities, it can be relatively easily switched with another or replaced with an internally customized component (37, s. 10). Modularity reduces cost of adaptation by shrinking the amount of skill and knowledge required to make changes.

Law of conservation of modularity

Open source is often a manifestation of Clayton Christensen's "law of conservation of modularity", that states that in a software stack, one layer is often modular and conformable to allow adjacent layers or modules to be optimized to achieve greater value (29)(41, s. 2).

> *Electronic Arts needed a fast, reliable server for a version of the Sims. Oracle proposed a Linux version of its application cluster software. To compete on the demanding project, Oracle delivered a competitive database solution by porting and optimizing its application cluster to run on commodity Linux servers. Here, Linux's modularity allowed Oracle to optimize its own, proprietary software and price its software at a higher margin, while still charging customers for less than a Sun Solaris based solution.*

Tradeoffs

Despite all the benefits, the choice of a modular architecture over an integrated, monolithic one is not a straightforward one. Modularity-integrality choice also tends to bring some tradeoffs. Modular systems are typically restricted in systemic innovation, where a change would require simultaneous change across the systems, both in how the systems operate, and how they interoperate (37, s. 16).

Another issue is that maintaining efficient modularity requires effort and discipline. If a system is developed by a small band of developers who all develop and know all parts of the system, maintaining modularity slows down development.

> *Apple's operating system OS X is fully developed by Apple's own developers as a proprietary application. This has allowed Apple to design and optimizations solutions that span through the entire software stack, which would be impossible or extremely hard to implement in a highly modular system. OS X running on an Apple laptop tends to have battery life 2-4 times longer than an Ubuntu Linux or Windows*[41] *running on the same machine. Speeds of suspend, wakeup and reconnecting WLAN after wake up are also multiples faster.*

[41] http://www.zdnet.com/the-other-hidden-cost-of-running-windows-on-a-mac-battery-life-7000000906/

Another example is the original IBM System/360 mainframe computer family, which had a non-decomposable architecture, in which every part effectively communicated with every other part. This made development highly complex, but offered unparalleled performance.

Netscape developers originally developed the browser with high integration to maximize product performance, given a dedicated team of developers and competitive environment, where quick to-market time was necessary (41, s. 27). Later on they changed to a more modular architecture to facilitate a community of developers.

3.4 Domain: Business

3.4.1 Applicability

Demand: Relative product importance and customer applicability

In general, the demand of an open source product can be analyzed in a two dimensional map, based on product's customer applicability and relative product importance (32, s. 14).

Customer applicability denotes the portion of market that can benefit from the product, whereas relative product importance is how important the product is to the user. An application designed for a rare operating system has a small applicability, but potentially a large relative importance. A screensaver on the other hand has high applicability, but low importance.

Many products that meet real customer needs are too specialized or too early to market to produce the return on investment that a large company or a venture-backed startup demands (32, s. 14), leaving unfilled market space for alternative solutions, such as open source products.

Products with high applicability and high importance, the “stars”, have most profit potential, but potentially have more commercial competition for the product and related services. These products are most likely to have large developer communities supporting them.

> *Linux falls into star category with its enormous user base and development force, and has garnered a flourishing and profitable industry around itself.*

Products with high applicability, but low importance are mainstream utilities everybody can benefit from, yet are not crucial.

> *TouchGraph's Google Browser translated search results into a graphical map. Though this may make for an interesting tool, it may not be commercially feasible itself. Still, such products may find more use as promotional items.*

Products with high importance, but low applicability are high profile nichers. These applications operate in small, yet critical niches.

> *SquirrelMail is an application Internet service providers can use to run their mail operation. This function is critical, but only relevant to a handful of companies. Due to criticality, customers may be willing to pay a premium for quality support and service, enabling commercial operations.*

Lowest segment is the one with low applicability and low importance. These serve a small niche, and scratch a small itch, leaving little for a profitable operations.

> *Wings3D is a powerful polygon mesh modeler, yet has use mostly among students of advanced mathematics.*

Adaptability

Commercial software companies rarely allow external developers to edit or even read their proprietary source code. This severely limits adaptability of such code base, as it cannot be customized to meet special requirements or fitted to new platforms. As open source code and documentation is readable to everyone, code can be fitted to needs or even ported easily to different programming languages entirely with minimal resources (1, s. 8).

> *Adaptability is a core reason Linux servers were used to render special effects to movies like Titanic and Lord of the Rings and operate such giants like Amazon, Disney, Dreamworks, Pixar and Google (60, s. 7).*

Presence of capable community

For a company to gain development support from the community, a community, or a potential for one, must exist. This imposes some limitations on how and what the company can develop. Niches, where experienced users may also be capable software developers have potential for a community. Without user-developers, little or no capable workforce is available to develop the project.

When assessing an existing open source community, company should look for size, talent and good organization (28, s. 25). Existence of a community itself is no guarantee of any help; instead a small, fanatic, disorganized, idealistic community may instead be a liability for commercial interests.

Alumni-effect

Open source code may already be familiar to programmers: because it is freely available to all, it can be used in schools and universities for learning purposes, thus creating an "alumni effect" (37, s. 9). Commercial software vendors often emulate this benefit through for example university licenses.

Existing standards

Most successful open source projects are general-purpose, commoditized systems infrastructure applications, where requirements are well known and APIs standardized (52). This limits requirements for common development activities, such as requirements analysis and interface specifications. Moreover an existing base of graduate students and hobby programmers may be willing to take part in projects like this.

More specialized industries such as medical systems, asset trading or image manipulation often have less technical professionals, who are less likely to develop their tools themselves. These domains require highly specialized technical skills and often exhibit little universal standardization.

Replication

Open source methodology is often incapable of effective user-interface innovation. However, there are many instances where open source has been successful in replicating existing, proprietary software. Even though innovation itself can be unfeasible goal, an existing application acts as an existing standard.

> *Harmony began as a project aiming to produce an open source version of then proprietary Qt-toolkit, a graphical user interface framework developed by Trolltech. Harmony project progressed so fast and so far, that Trolltech eventually had to release Qt as open source to combat the threat and to stop the Harmony project.*

Dominant competition

Choices about business strategy are not always about optimal growth. For smaller software developers in fields with strong network externalities, there may be no space left to live in behind the incumbent. When faced with almost certain doom, companies may still have a fighting chance of their existence by turning into open source (38, s. 225) and cultivating a community in the shadow of the colossus.

Open source projects sometimes tend to gain momentum when competing against a dominant company (38, s. 228). This effect can be interpreted in many ways. Existence of a single strong incumbent is often a sign of higher prices in the market, which often leaves more room for cheaper, lower tier competitors. Furthermore when multiple smaller actors compete against a giant, competition between them is often more akin to coopetition than direct competition, as the best way to fight an uphill battle is to stay united.

Incremental changes

Open source development by nature tends to consist of individual incremental changes and commits (11). Historically this has meant thriving in areas where incremental change is rewarded, which has meant back-end systems more than front-ends, as back-end systems are often less vulnerable to technological disruption and fads.

Evolution of software can be thought to be divided roughly in two categories, phylogenic and ontogenic (1, s. 5). The former refers to the evolution of a species, whereas the latter refers to the evolution of individuals. Open source development is more suited to ontogenic development, whereas phylogenic changes (larger, systemic, architectural) require more detailed, in-depth planning and coordinated execution, more suitable for traditional software development.

As projects mature, issues like backwards compatibility and synchronization of upgrades must be confronted (38, s. 228). Maintaining full backwards compatibility requires a lot of resources, which may be hard to attain from an open source community, as there is little bliss in working on temporary solutions for non-core users, as is the case in backwards compatibility; technical users are expected to upgrade their system themselves, even if it requires customized changes in code.

Embedded software

Embedded devices often have severe limitations in computational capacity, memory, disk I/O and battery lifetime. Also they often incorporate extremely non-standard hardware functionalities that require special functionalities from the software, such as near real-time operations (e.g. audio equipment) or almost mathematical reliability (e.g. industrial controlling systems) (28, s. 62). These factors make open source software an intriguing option for developers of embedded devices, as it gives them all options to fine tune functionalities and performance of their system. This is an example of “Law of modularity”.

A recent example of an embedded open source application is Android and its march of success in mobile phones. Android is an open source mobile phone operating system developed by Google and based on Linux kernel. By commoditizing lower levels, Google has moved value up the software stack e.g. with their open Android market, where all Android users may purchase applications.

Platforms for new ecosystems

When developing software for professional developers, it is often more feasible to build the system as a general framework with included and documented interfaces and embedded scripting languages, instead of a monolithic, standalone application. Developers tend to tinker and craft their own modifications and extensions, potentially creating an ecosystem around the platform (11, s. 4). Many open source projects have aimed for this platform effect.

IBM's Eclipse project allows other companies to develop plug-ins and extensions for Eclipse, which can then be purchased in Eclipse's marketplace. Canonical's Linux based Ubuntu desktop environment also has incorporated proprietary software into it's market place.

Roxen is a commercial company that develops two complementary products, an open source web server, and a proprietary content management system (CMS) that runs on the server. Web server is under GPL, whereas the CMS is under proprietary license. Company has nurtured a community around the server, to help with development efforts.

Hosting

Open source licenses require no actions or prevent no modifications, should the modifications be used internally, without distributing them. Commercial companies are basically free to use open source software as they wish, as long as software itself never leaves their servers. Most notably web companies, such as Salesforce, eBay and Google benefit from this detail, as their customers simply interact with their servers via HTTP-protocol (29, s. 1).

Technology choices

Attractiveness of an open source project for potential open source contributors depends greatly on the project's technology choices (59, s. 1222). To gain attractiveness, a project must utilize a well-known, fashionable or otherwise accepted programming language and framework. Programming fashion is somewhat unpredictable, and changes over time. For example Java has long been somewhat considered a “necessary evil” due its overt verbosity and old age, but still has an extremely strong presence[42] in large organizations. However, Java developers tend to be older, paid professionals and have less interest in developing it on their free time.

Governments

Many large institutions, especially governments, that are extremely reliant on their software infrastructure, have specialized requirements and value self-reliance highly. For many governments, such as India and China, dependence on foreign software vendors is highly appalling, especially in the post-Snowden world.

Open source offers a solution to most of their issues by allowing code to be modified, inspected and developed internally without any need for foreign third party vendors with questionable trustworthiness. Over time, China has produced several Linux-based operating systems for their internal use, such as Kylin and Red Flag Linux. India and China have also invested heavily in education of software engineers, slowly increasing their significance as software super states (30, s. 31-32).

High confidence

For many safety critical domains, such as nuclear power control and flight control, there often exist strict, legal requirements regarding documentation, development method, testing, quality assurance and other processes. For many open source projects, passing these rigorous qualification certifications can be structurally impossible, as company cannot always be 100% certain which individuals have actually contributed the project. Nevertheless, the issue is not clearly cut; nuclear control and flight control include tall stacks of systems with multiple functions, and not every single one is strictly regulated.

End-user applications

Compared to server side software, end user and desktop applications are hard to write, as the programmer has to deal with a constantly changing graphical windowed environment, with no standards to lean on. In addition, end users often consider quality graphical user interface even more important than the technical quality and functionalities of the application itself (11).

[42] http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

High competition

Even though open source can effectively compete with price, it often lacks in highly sophisticated, groundbreaking functionalities. Customers in domains such as on-line trading systems are willing to pay heavy premiums for higher quality, faster development and advanced features, leaving the domain unsuitable for open source development. In addition to smaller markets, every developer specialized in on-line trading is likely to make a killing by working for proprietary companies in these domains.

Developer tools

As open source thrives in areas with heavy usage, where overlap between developers and users is high, developer tools appear as a prime candidate. In addition to code editors, tools such as compilers, debuggers, configuration managers, bug trackers, memory leak tools and performance profilers exist in this same user-developer space (52).

> *Many successful text editors and IDE's are licensed as open source, including Vim, Eclipse, Emacs, Anjuta and Aptana Studio*

Niches

Certain technological communities, such as scientific community and partly even the defense industry, have software needs that are unappealing to commercial actors. Areas that might be too small, too specialized, seemingly temporary or otherwise hard to access, leaving more room for open source solutions, which may grow slowly to fill these niches.

> *Linux-based Beowulf clusters were developed in the scientific community partly due community's non-standard hardware, which was not supported by the mass-market software houses (52).*

3.4.2 Appropriability

A company's ability to capture profits value from a project is called appropriability. Appropriability regime is said to be “weak” when the used asset is hard to protect and “strong”, when it is relatively easy. In case of open source, the appropriability regime is often weak by nature, and extracting premium from software licenses impossible. Appropriating value from knowledge, experience, expertise and information is more difficult, as unlike software code and ownership, they are non-rivalrous in usage and harder to deny access to (41). In case of weak appropriability, companies must rely more on speed to market, timing and luck.

In the software world in general, value capture is hard, while value creation is easy. This is as true for open source companies as it is for consumer web service providers such as Twitter. Historically, even technologically and socially successful open source projects have had hard time generating profits (53). Despite open source communities being often powered by pro bono work, commercial companies are ultimately driven by financial benefit.

Types of value

Direct cash flow is not the only vector of appropriating value; other, indirect vectors support business and enable improved cash flow in the future.

Cash

Even though open source can seldom be sold for a premium, money can still be appropriated through e.g. proprietary add-ons, dual licensing, supplementary hardware and consultation services.

User base and adoption

Open source enables companies to gain large user bases with relatively little investment in marketing and distribution (32)(10, s. 3). Though users don't directly generate cash flow, they are prospects for future sales of software and consultation. Users also help development through feedback and marketing by promoting helpful software at grass root level.

Product quality

Open source community and other open source components can be utilized to improve the quality and features of company's own products. Better products improve customer experience and help with sales (25, s. 47).

Complements

Vivid developer community can often develop add-ons and other related software around an open source product, without any action required from the commercial company itself. Complements increase the value of the software ecosystem, thus increasing perceived value of the software itself (9). Other similar vectors for appropriation include market positioning, lead times, network effects and first-mover advantages.

Market feasibility

For financial success, producing a successful and widely adopted application itself does not help, if users either don’t want to modify it or won’t buy any additional proprietary components or services. For a company to tap into the cash flow, it is required that the customer markets exhibit some special criteria. E.g. A segment exists that will pay for proprietary license or services, or that the company has some additional component that complements the open source offering, and can be sold for money.

> *An example of a complementary, proprietary component based on open source is Mac OS X, where Darwin, the operating system core itself, is open source, but the graphical user-interface is not.*

Small company leverage

Incumbents benefit from their large market shares, but may find difficulties in facing the open source wave with their existing business methods. They tend to suffer from the “comfortable clone syndrome” (4): the loyalty to “the way we do things here” (41) reduces the ability to envision alternative possibilities and go beyond the established business models.

While proprietary software requires big teams to internally develop and support the product, what matters in open source initiatives is the community size, not the corporate size. As the community manages many activities regarding quality assurance and development, higher quality products can be developed faster, allowing opportunities for small, lean and agile companies (49).

Wide slice of value chain

Traditionally software companies have pricing power and can extract reliable revenue through licenses alone, offering related services and complements as a bonus. Open source companies are often obstructed from this source of revenue and rely on these adjunct services (30, s. 10), where their pricing power is not exclusive (2) and they compete on the basis of customer service, instead of intellectual property ownership. As a result, these companies often tend to do business in a wide slice of the whole value chain.

Winning adoption

Nothing is easier to distribute than free software. Open source software has historically good track record in gaining market share in installed bases globally, since there are no issues of taxation, licensing or piracy (32). Larger installed base can offer a company various benefits, such as development support, marketing value, brand, potential clientele and a safer market position.

However, a notable issue is that this gained market share, though easily large, tends to be the most price sensitive lower segment of the whole market, whereas most money can be found in the upper segments. For example MySQL is by far world's most popular database by installed base, but almost nonexistent when measured by revenue and compared to proprietary database providers.

If a project is successful enough, it can gain a status of a de facto standard in it's domain, with even potential competitors joining in, leaving the industry to compete in company's own turf (4, s. 19)(29, s. 1). Google has strongly advanced developments in the HTML5 standard, including developing and opening a specialized video encoding format. As Google has strong presence in the web, the more applications and services utilize HTML instead of competitors such as Flash or Silverlight, the stronger their position. Other similar de facto platforms include Linux and Apache (60, s. 7).

Piracy to promotion

Unlike many other specialized fields, software environment includes a great number of hobbyists, young people and non-commercial actors. This combined with high price of specialized software and ease of illegal distribution of software creates a favorable environment for illegal piracy.

Traditionally software companies have considered pirates a threat and persecuted them, often gaining negative PR in the process. This tradeoff is often rational in business sense, as proprietary companies have license revenues to defend. Open source companies by definition don't suffer from piracy, as all these users are considered valuable, since they would have not paid the license fee anyway. This allows utilizing alumni-effect to grow the user base.

Maximizing returns to internal innovation

Companies require a wide range of approaches to maximize the returns on internal innovation, as not every innovation is suitable for feeding to the product line. Other options for intellectual property licensing, patent pooling and giving away software for free may help to stimulate demand of other products. Methods for benefiting from internal innovation include:

- Generate intellectual property for internal use (Proprietary model).
- Build absorptive capability and identify external sources for intellectual property.
- Generate intellectual property for external use, through patent portfolios.
- Generate intellectual property for indirect benefit through spillovers.

Open source provides a suitable method for spinning out projects that would otherwise lack any strategic value (65).

Reputation

The open source phenomenon garners lively interest not only among scholars, but also in the public opinion. Taking part in open source projects may therefore be used as a strategy to polish the corporate image. Working with an emergent technology is also a good way to gain the interest of venture capitalists (4, s. 18).

Hiring

The open source community is a meritocratic process, where individual's skills can be easily identified and verified (4, s. 16). For a company working closely with an open source project, related community provides an easy and reliable pool for talent (25, s. 44).

> *The norwegian software company TrollTech, producer of open source development framework Qt, has hired most of its 230 employees from Qt's community.*

Defensible market positioning

It is hard to compete with a free product. Open source business models allow companies to leverage this to enable highly defensible market positions. In exchange, companies forfeit some of the potential for premium revenues.

A prime example of maintaining a defensible market position is Cygnus, the company developing CGG, a GPL-licensed compiler. Every year, Cygnus makes revenue by porting GCC to various types of hardware, and maintaining those ports. All changes are licensed as GPL, and therefore available for free. Cygnus only charges for the expertise and effort involved in the port and maintenance, not for the code itself. Even though any competitor can simply take GCC source code and start a business, developing and maintaining GCC is a highly knowledge and experience-dependent process with very high costs to the customer in case of errors. A competitor would also be required to release their code under GPL, denying them any permanent competitive edge through technology alone. As porting of GCC is a rare and relatively cheap, yet crucial, event for Cygnus' customers, they are ready to pay a premium for top of the line service.

Life straw

If a company is lagging behind the market leader and losing money, open sourcing the software may allow the company to extract some value (37, s. 13). Moreover potential customers may be skeptical of how long the company will maintain its software; it is not uncommon for vendors to suddenly discontinue a product line, leaving old customers alone with their obsolete software. Open sourcing may reassure customers that the software will never be withdrawn, and someone (or themselves) is able to further maintain and develop it.

In 1998, Netscape open sourced a portion of its web browser, Mozilla. Back then, the company was in a downward spiral and fully open sourcing their software allowed the project to live on and eventually transform into the Firefox browser still used today.

Patronage

Though many benefits from open source development fall on small companies, large for-profit companies can also utilize it strategically, by founding and supporting projects with desirable direction and technology. This way a large company can subtly influence the development of the technological landscape behind the scenes, with minimal exposure and little monetary cost.

Spinouts

Software laboratories and developers sometimes produce software they cannot commercialize effectively. For example development and maintenance tools often are susceptible to poor commercialization vectors. Open sourcing tools like this enables the company to gain goodwill from the community, as well as gain support in developing these tools for their internal requirements with little or no cost. Also, as the company still has supreme knowledge on the project, should a sufficient user base build up, the company can attempt to capitalize on this by providing education, plug-ins, services or other complementaries.

3.4.3 Services

Multiple studies claim that enterprise solution fees typically consist of 30% license and 70% implementation and maintenance (29) (30, s. 29). Moreover, according to a 2000 US Department of Commerce study, packaged software licenses have not topped 30% of total software investment since 1962 (29). This ratio holds true for proprietary solutions. For open source solutions, value-adding services is the basis of revenue. It is critical for software companies, and especially for open source companies, to consider themselves to be in the service industry, instead of the old fashioned manufacturing industry (53), as sale of licenses alone is often insufficient to sustain commercial software business (32, s. 3).

As software development, and therefore copying, is relatively easy and fast, sustainable competitive advantage cannot be achieved by adding features. Instead, organizational innovations can be utilized to produce services in a scalable fashion, which is much harder for a competitor to copy (41). Provisioning may include services such as consulting, system implementation and integration, support, maintenance, remote administration, training and application management (2, s. 4). Providing services like these does not rely on economies of scale, and therefore allows viable business for small and medium sized companies.

For customers the bottom line is that paid service generally equates to higher quality services (32, s. 10). Third party service providers can often be found locally and hence can provide on-site assistance.

> *Every company that turns to Linux for their servers is a potential customer for IBM's services. However, the change from another platform, such as windows IIS, is extremely difficult and expensive even without the licensing costs, so open source provides a good additional lure.*

Service platform

Open source projects offer companies a low cost and low entry-barrier environment for developing their services and building service related businesses, such as education, customization and maintenance. Building on an existing code base saves a company resources and allows them to leverage the project's existing installed base.

Certification

The software world has a deep history of incredible hypes and corresponding busts; new technologies have failed to deliver and unsuccessful products have been withdrawn from markets. With a history like this, customer skepticism with new trends and technologies is understandable even with proprietary software. In addition to general software unreliability, open source software has the aspect of hazy ownerships and new legal practices behind it.

Commercial companies may remedy skepticism by offering certification programs for other companies working with given open source application. Certification of a generally trusted company is a signal of reliability and stability to the customer.

> *Linux certification programs from Linux companies, such as Red Hat and Novell, have greatly reduced support concerns previously held by customers (29).*

Up-selling

For many software sellers, gaining new customers is often a tedious effort. Instead, companies can benefit more by up-selling, i.e. providing existing customers with new solutions and services on top of previous ones. As license sales would require customer to purchase new software, selling services is a logical step for up-selling (32, s. 3).

Support

As open source products grow in popularity, demand for quality support also grows stronger (32, s. 2). In addition when software companies mature, relative importance of licenses grows smaller, and similarly importance of services such as support grows larger. One estimate is that by the 20-year mark, a typical software company will have \$2 of service revenue for every \$1 of licenses (29, s. 1). This underscores the importance of the paradigm shift towards services in software industry.

A problem might arise if customers tend to have modified versions of the software. This may increase resource requirements for effective support services, in the form of more skilled personnel and detailed processes (25, s. 47). However, use of customized software correlates with the customer's own technical merits, and technically adept customers are more likely to find support from the community (25, s. 47). In addition, the support provider has the option to contract external, specialized developers to assist their customer in more technical details.

Consulting

In software world, it is not unusual to find many similar applications for the same purpose. These applications may vary greatly, each with its own strengths, drawbacks, versioning systems and release cycle. For non-technical customers, selecting a software and version is can a daunting task. A consulting company with experience with a given software may provide the customer with assistance in selecting a product fit for the customer's needs (32, s. 10).

Integration

Software systems often form an environment where multiple integrated applications work together. Whether or not multiple given applications work together on their own is another question. Companies may provide customers with value by developing integration mechanisms for multiple software products and offering the package as a whole, ensuring that a suite of applications works well together (32, s. 10).

> *Cognizant is an example of an integrator, which has removed nearly all licensing costs from their solution proposals to create winning bids for customers, achieving both lower prices and higher margins. The company has a large development center in India, which powers its integration projects around the world (29).*

Hosting / Web Services

Open source license restrictions require disclosure of source code, in case the code is distributed in specific conditions. However, in internal use, open source software places no limitation, as the code is not distributed at all. Companies providing web services, such as Amazon, Salesforce.com, eBay and Google, can utilize open source in their operations in any form they desire, as customers only access the HTML data generated by their backend systems. In essence, a company is allowed to make a value-added web interface for an open source application and sell access to it without any licensing issues.

Even though open source licenses do not affect web service hosting in any way, many successful web service providers have adapted policies ideologically not far from open source; many of them, e.g. Amazon and Google, treat their web service API's as programmable components, allowing external developers to integrate with their internal systems by reading and writing data.

This “Open Surface” allows a web service provider to enjoy some benefits typical for open source software. As more developers grow accustomed to a given API, they form a community, which provides other users more value in the form of support and complementary software, adding value to the web service and therefore benefiting the service provider.

3.4.4 Market dynamics and competition

For commercial companies, open source is not an end itself, as technology or ideology themselves don’t pay the bills. It is merely a means to an end; a tool for building sustainable business through innovation, imitation or both.

Revenue streams

Revenue streams of open source companies usually consist on some or multiple of the following (25, s. 49)(33, s. 102).

- Support Selling
    - Revenue from media distribution, branding, training, consulting, custom development, and post-sales support.
    - Red Hat, enterprise Linux distribution
- Loss Leader (Dual licensing)
    - A freemium open source product is used as a loss leader for proprietary software
    - Phusion Passenger, web server
    - MySQL, database engine

- Widget Frosting
    - Selling hardware that is tied to open source software, such as drivers and interface code.
    - Raspberry Pi, miniature computer scaffold
- Accessorizing
    - Selling books, computer hardware and other physical items associated with open source software.
    - HP, promotes open source in areas where it sells hardware
    - O'Reilly publishing, has published a great deal of books of open source software
- Web service Enabler
    - Open source software is created and distributed to support access to revenue generating online services.
    - Github, web service code repository
- Brand Licensing
    - Charging other companies for the right to use its brand and trademarks in derivative products.

Loss Leader

Integrated development environments (IDEs) have often utilized the loss leader approach successfully. Traditionally IDEs have been expensive and complex proprietary applications that provided steady revenues if they managed to attract a large user base.

> *IBM chose to turn its Eclipse IDE into open source, despite the source code being then valued at 40 million dollars. The choice substantially increased the popularity of Eclipse and expanded the market for complementary products. Since then, several companies have followed suit, Sun with Netbeans and BEA with Beehive, both of which eventually were acquired by Oracle.*

Physical install medias

A survey of 113,794 Linux users indicated that 37% of respondents preferred to obtain their Linux distribution as a CD/DVD. According to distrowatch.com, as of February 2003, the highest price charged for a Linux CD was 129 dollars. However, this data is old, and Internet connection speeds have increased in magnitudes since then, lowering the perceived cost of downloading software.

Upgrade Services

Active open source community often develops the application relatively fast and in short increments. However, not all of these increments or versions are completely backwards compatible, nor provide features the customer actually requires. Upgrading software is always a risk due multiple factors that may go wrong and even require a rollback. Open source distributors have deeper understanding of their software and can provide customers with the latest, necessary upgrades seamlessly.

Dual licensing

MySQL is a common example of dual licensing strategy, which is most akin to the loss leader model. Millions of free copies of MySQL software have been downloaded, but only about one customer in one thousand has purchased a commercial license (15, s. 592). The proportion is small, but amounts to thousands of paying customers. Combined with low cost open source development process, this has turned into a sustainable business. It must be noted that MySQL was acquired by Sun and hence is today owned by, again, Oracle. However, a competing open source project, MariaDB, has taken MySQL as a base and hijacked the development. Other dual strategies include Red Hat with free Fedora and RHEL (Red Hat Enterprise Linux), the latter of which contains proprietary software, and Sun with Staroffice and Openoffice, which, again, were acquired by Oracle.

Underdog leverage and entrenching

Companies often resort to combining open source and proprietary software when facing large incumbent firms such as IBM or HP (2, s. 1). Open source often acts and spreads via channels not reachable by traditional sales and marketing efforts, providing disproportionally more leverage to the underdogs. As a result, many of the incumbent have released their source code as open source, being required to play with the same rules as everyone else or face irrelevance (25, s. 47).

A large installed base and network externalities can easily provide a company with a strong, defensible market position safe from even predatory pricing. The more reciprocal the utilized license, the harder it is for competitors, and the company itself, to profit from the software (54).

> *Trolltech is a software company developing a wide spread graphical user interface toolkit called Qt. Originally Qt was released under a proprietary license. However, the developer community disliked the proprietary license and started a project to develop an API-compatible version of Qt, under GPL. This project, called Harmony, gained ground and eventually advanced so fast and so far, that it forced Trolltech to release Qt under GPL to mitigate the threat of becoming redundant.*

Cracking entrenched markets

As open source can be used to create heavily entrenched positions, it can also used to disrupt them. By successfully commoditizing a layer of the software stack, companies can eliminate appropriability vectors from competitors dependent on that layer.

> *For example, IBM has used Linux to crack Microsoft's dominance by eliminating server fees from Windows (29, s. 1). Also, IBM's code editor Eclipse was valued at $40 million*

when IBM open sourced it. Eclipse grew extremely popular and shifted value up the software stack, leaving less space for Microsoft's and Sun's proprietary tools.

Public interest

For public organizations, open source provides a technological platform free of foreign company policies and private interests. Some governments have experimented with moving to an open source model.

Lately, Munich ditched Microsoft products for desktop Linux operating systems, customized to match their preferences. In 2013 Argentina launched its own Linux version called Huayra *to be used on government laptops and in schools. Venezuela has a similar pattern with its own distribution, Canaima. China also has had a long history of experimenting with its own versions, including Kylin and Red Flag Linux distributions (32, s. 3). Especially after recent NSA leaks, many governments have lost trust in commercial solutions from USA based companies.*

For developing countries, open source is a way to gain working solutions with a fraction of the price of commercial solutions. A Windows operating system with Microsoft Office is worth of about 1 month of per capita income in Vietnam.

Many governments have proposed and occasionally implemented a variety of measures to encourage open source development (37, s. 21). In USA, the President's Information Technology Advisory Committee recommended federal subsidies to open source projects that advanced high end computing. The European Commission also discussed supporting open source developers and standards.

Market destruction

Open source bases its effective propagation and success mostly to the lack of price. This feature brings about a striking change in markets where the open source gains ground; markets get wrecked and the total turnover shrinks radically.

Companies changing from commercial database solutions to open source alternatives, such as MySQL, effectively stop paying any service provider, or at least reduce the cash flow significantly. There are estimates that MySQL has achieved about 40% of database installed base, but only amounts to about 0.02% of the dollar turnover.

Coopetition

Companies often utilize open source as a business enabler, as it does not generate turnover by itself. This, combined with the open source often starting as a disruptive underdog technology, translates to a curious competitive environment, where seemingly competing companies actually benefit from the success of each others; hence the term, coopetition. In this situation, improving a common open source product is a direct spillover to direct competitors, but still economically feasible (63, s. 9). Moreover in open source business, appropriating internal innovations is often unfeasible or impossible to start with, though cross-competitor coordination would likely be impossible without open source procedures to begin with.

For Linux distributors, the worst competition is not other distributors, but non-Linux operating system solutions, such as Windows and OS X, Linux's technical performance and usability, and public awareness of the Linux. All efforts that wrestle markets from incumbents also improve the chance any other company might succeed in it afterwards. Effectively, companies complement each other in creating markets, but compete in dividing the markets (32, s. 18). As long as there is any of room to create new markets, inter-company competition is unlikely to gain foothold (63, s. 6).

> *United Linux is a consortium of several Linux distributors that joined forces to minimize development overlap by coordinating engineering efforts. Participants seemingly compete with each other, but still gain a net benefit from coordinated efforts. Also, United Linux acts as an effort of smaller distributors against the incumbent Linux provider, Red Hat.*
>
> *When introducing Java, IBM was not aiming to make money off the platform per se, but attempted to gain adoption for it to help them develop and distribute other value adding solutions higher up the software stack (63, s. 12).*

Weak lock-in

Compared to proprietary software companies, companies providing open source solutions have significantly weaker lock-in, as any one inside or outside the customer company is allowed to edit the software (61). This weaker lock in reduces premiums the company could earn.

Inter-company collaboration method

Software development over cross-organizational boundaries is often hard to accomplish due many practical issues, such as code ownership, trust, licensing, and organizational command structures. Open source projects enable commercial companies to collaborate with well-defined and understood rules and with no fear of hi-jacking.

Open source enables efficient development and change of ideas around the core project, even when participants (competitors, consumers, public institutions, universities) themselves are plagued with mixed incentives and internal rigidities.

This effect on trust is especially remarkable in smaller companies that are often uncomfortable doing business with large partners due their lack of leverage and negotiation power (38, s. 225). These barriers can sometimes be managed by for example utilizing open source practices, with smaller developers contributing to an open source project against monetary benefits.

Venture capital

Development of successful open source software has high signaling value also for institutional investors. As work is conducted with minimal monetary compensations, a successful project is almost a sure-fire sign of existing leadership and technical capabilities. Founders of companies such as Sun, Netscape and Red Hat have signaled their talent for business in open source projects (27).

Sales pull

A common change in open source world is the move from sales push to sales pull, where marketing and buzz keep interested customers swarming to the company's website to download the software for evaluation purposes. If the software is deemed worthy and a customer decides to adapt it, they contact the company for pricing information and services. This is a striking difference compared to the common sales push method, where sellers contact customers. Similar pattern are also known as "Seed and harvest – sales".

Funambol described its customer acquisition process for its Syn4j product as such

1. *The potential user reaches the Sync4j website to collect product information and technical documentation.*
2. *The product is downloaded.*
3. *After downloading the product, the potential customer often subscribes to the Sync4j mailing list to receive free initial support and guidance.*
4. *After intensive use of the software (usually in R&D projects), the user contacts Funambol asking for price information and license conditions. Internally, the customer is classified as a Prospect Customer.*
5. *After being informed about general sales terms, the prospect requires a formal commercial offer (becoming a Lead) and, if it fits with its expectations and budget, it becomes a Funambol Customer.*

Compared to push sales, the sales pull effect is drastically less efficient, requiring large user bases and loads of time for sales cycles. However, sales pull requires significantly less, or even no sales personnel at all, providing a sustainable option for small companies with little resources. Also, for small companies with little brand value or customer awareness, direct sales is not always an option to begin with; addressing large companies as a start-up can be a purely impossible task.

Phusion, company developing Phusion Passenger application server utilizes this model; Phusion Passenger is open source software, which can be easily obtained and used without restrictions. However, Phusion also offers a more sophisticated version of Passenger, which includes optional features more valuable to commercial companies, which then contact Phusion for purchase.

4 Customer perspective

We have discussed in depth about how open source software manifests itself on the provider and developer side, upstream of the value chain. Although the mechanical differences are most significant on upstream, understanding customer and end user perspective is necessary for successful sales and business development.

Most common fears regarding usage of open source relate to (25)

- Documentation availability.
- Quality of documentation.
- Maturity of the product.

While most common perceived benefits in using open source are

- Quality, technical merits and ability to meet requirements.
- Free availability and low risks.
- Lower total cost of ownership.
- Opportunities for innovations.
- Independence on software vendor.

Usability

As open source development has not proven to be an excellent methodology for producing good user interfaces for desktop use. As most users of the experimental software are highly technical with high tolerances and experience, requirements and modifications are based on this input, leaving non-professional users with poor interfaces. Weak usability incurs costs as reduced employee performance.

Vendor independence

As open source licenses leave vendors with little lock-in power, customers are left with more negotiation power. Should a vendor turn out bankrupt, incompetent or expensive, there are no licensing restrictions barring the customer from changing the vendor (28, s. 13). Companies are more self-reliant with the option of source code modification, allowing them incremental project and upgrade schedules and free reign on integration decisions (29, s. 1). Also, companies are free to co-operate with the community directly, should they have a need and resources for it. Open source allows customers to implement projects based on their own requirements, and not the goals of a third party provider.

Cheaper services

Because open source service providers compete against each others to provide support for the same software, they have less leverage to raise high premiums (28, s. 13), as service provision is highly susceptible to market forces and competition. Services such as support are often, to some extent, provided by the community for free (32, s. 11). This however requires the customer to have a necessary understanding of the technical context. Given technically adept employees, the community can often offer high quality support service for the user.

Service for proprietary software can often only be purchased from the software developer. Considering high costs of changing software, this gives the provider high pricing power.

Government

Vendor independence and lower prices are central issues for public organizations and governments. Especially after recent NSA spying episodes, governments distrust proprietary service providers, whose code they cannot inspect. Public institutions also tend to be quite large and costs in services and licenses add up fast.

The US government, including Department of Defense, Department of Energy and NSA utilize open source software. The national, state, and municipal governments of Germany, Peru and Chine are considering, and in some cases requiring, the use of open source for government applications (60, s. 7).

Version proliferation

Unlike commercial companies, open source projects base their release schedules and versions on development requirements, not on user preferences. This often results in a fast release cycle with multiple versions, where choosing a version or choice to update to a newer version may require technical understanding (32, s. 12).

Future-proofing

When adopting an infrastructural system, companies want to mitigate future risks, such as the need to port the system to a new platform. One central future-proofing issue is the option to develop the system onwards, implementing future protocols, integrating it with future systems etc. It's often not sensible to only rely on the original software provider for future support, as it might be far too costly, or the company might be acquired or goes bankrupt, or the product discontinued.

Most proprietary licenses forbid changing the program code, nor do they provide the original source code with the product to begin with. Both of these, legal permission and source code access, are needed for implementing changes (28, s. 13). Implementing an open source license is a guarantee for the customer, that the system can be changed afterwards with no hidden fees or landmines.

> *One example of infrastructural software gaining momentum as open source is the cloud-computing platform with entrants such as Openstack and Eucalyptus, both being supported by big software powerhouses and developed for their needs.*

Compatibility

Another issue with inertia is the classical format war, which has prevailed for ages in one form or another. For example, in large organizations, interoperability of formats across departments and customers is important, yet not always obvious. There are also issues with future-proofing the system, should one provider fail.

Especially in public organizations like governments, dependency on private, foreign companies is often quite undesirable, and as a result, open document formats and office tools like Open Office have been recently gaining momentum.

High quality

Software being open source is not a quality certificate for any code base. However, it has been proven that well functioning open source communities and development methodologies provide high quality code, even when compared commercial competitors. Especially the speed of bug fixes is often in another magnitude compared to large software providers.

Support

Despite often having often no formal support channels, open source communities have a record of offering users high quality support for free. Usenet, mailing lists and forums can provide clients high quality technical support fast (34, s. 938). Though it goes without saying that as with quality of open source, open source is not a guarantee of quality support, but open source communities may still be are able provide it. Customer support for communities around Apache and Linux have even won awards of excellence (32, s. 4).

Documentation

The issue of documentation in open source communities can be quite debatable, as the definition of “good” documentation varies based on the background and skills of the one requiring that documentation. Open source software is often developed by highly technical people for other highly technical people, with documentation matching this environment. A company employing technical professionals may find mailing lists, forums and man-pages invaluable, whereas less technical companies might find the same material completely useless.

The open source community enjoys well-aligned motivations when it comes to writing documentation; should developers wish for fame and glory, they need to provide sufficient documentation for their code. However, they are not rewarded for providing superfluous novels of unnecessary documentation, which may be the result in commercial companies, where technical writers have skewed incentives.

Adaptability

Well-written and well-configured open source software can be easily ported to a variety of heterogeneous operating system and compiler platforms. In addition, since the source is available, end-users have the freedom to modify and adapt their source base readily to fix bugs quickly or to respond to new market opportunities with greater agility.

Reduced software acquisition costs

Open source software is distributed without development or run-time license fees, though many open source companies do charge for technical support. This pricing model is particularly attractive to application developers working in highly commoditized markets where profits are driven by marginal costs. Moreover, open source projects typically use low-cost distribution channels, such as the Internet, so that users can access source code, examples, regression tests, and development information cheaply and rapidly.

Total cost of ownership

When considering costs of software, there are many components to take into account, such as upfront and running licensing fees, cost of education and services, cost of customization etc. For mature companies with mature software infrastructure, cost of licenses is usually a fraction of the costs of services. One estimate is that for large companies, hardware and software together comprise less than 20% of the total costs of corporate computer ownership (3, s. 65).

Utilizing open source software is shown to reduce costs of licenses and services up to 20 percent (28, s. 65). The exact number varies wildly between companies of different industries, sizes and capabilities. Companies with internal technical expertize can capitalize on their skills and save considerable amounts of money, whereas less technical companies are still bound to purchase customization and services from commercial providers. Even without internal competence, companies purchasing external services can capitalize on open source environment's healthy competition. Competition with open source software has also reduced prices of commercial companies, as IBM and Sun offer their previously expensive Unix variants practically for free (28, s. 65).

It is reasonable to expect that utilization of open source software also incurs costs due to the lack of existing expertise and issues in interoperability with proprietary software. Especially in companies with high employee turnover, cost of ownership can be as much as 50% higher than in more stable companies due to the increased cost of training and reconfiguration (6).

When considering the total cost of ownership of a software solution, it is necessary to form a breakdown of total cost of ownership based on following factors (28, s. 16)

- Direct Costs
  - Software and Hardware
    - Software
      - Purchase price
      - Upgrades and additions
      - Intellectual property/licensing fees

        - Hardware
            - Purchase price
            - Upgrades and additions
- Support Costs
    - Internal
        - Installation and set-up
        - Maintenance
        - Troubleshooting
        - Support tools (e.g., books, publications)
    - External
        - Installation and set-up
        - Maintenance
        - Troubleshooting
- Staffing Costs
    - Project management
    - Systems engineering/development
    - Systems administration
        - Vendor management
    - Other administration
        - Purchasing
        - Other
    - Training
- De-installation and Disposal
- Indirect Costs
- Support Costs
    - Peer support
    - Casual learning
    - Formal training
    - Application development
    - Futz factor
- Downtime

Performance

Especially for companies relying heavily on computer infrastructure, such as Google, Amazon or Yahoo, software performance is essential. Open source software has been making its way into large companies such as these due to its often superior performance in specific areas, and its capacity for optimization (32, s. 17).

Performance itself is not a trivial measure either, as it depends on the use case and how software is configured to match its purpose. Due to the code being modifiable, open source software such as MySQL, Linux or Apache can be easily modified to fit the given purpose.

Lack of natural accountability

When company purchases software from a commercial vendor, that vendor is held accountable for the given product by law. If the product does not work as expected, the vendor can be asked to fix it, or sued for contract breach.

If a company uses open source software, there is by default no party to be held accountable. Most open source licenses contain waiver clauses exempting the owner and the distributor from all legal requirements, as software is distributed “as is”. Due to the concept of licenses and ownership, open source community is structurally unable to provide legally valid guarantees of software quality. Even if the software itself has been proven to be of high quality, many companies, especially larger ones, may find the lack of an accountable party unsettling.

Many commercial companies have stepped up to address this issue by offering guarantees for a charge (25, s. 44). Companies such as Red Hat, Canonical and Novell function as legal entities their customers can rely on, providing customers help with their Linux systems.

5 Conclusions

Open source is not a business model. It is a licensing scheme and social phenomenon that can be used in many different industries, from hardware manufacturing to software development and consultation. Furthermore it can be used in many ways; open source software can be developed, distributed, supported, marketed or used as a component in a proprietary application. Each of these ways can be used in a different business model.

Open source provides a fundamentally new method of developing software, by utilizing crowdsourcing to improve development efforts, while lowering costs. Small companies may use open source methodology to gain significant presence despite having inferior resources compared to larger competitors. In addition, open source provides a framework that allows commercial, competing companies to align their incentives and to collectively develop software they all benefit from.

Large companies may employ open source for other strategical maneuvers, for example by utilizing software that could not be sold for profit, or by indirectly influencing the development of a certain technology or industry.

Open source is not a method for maintaining superior profits, but instead enables low-cost business models with high sustainability and defensibility. It can be used as a central component in profitable software business, but requires other channels for income, as license sales are often not enough due to severely limited appropriability.

Choosing the path of open source licensing software is not without risks and drawbacks. These include market destruction through previously paying clients choosing the free alternative, loss of appropriability through licenses, risk of losing control of the software and its development path, and risks of legal litigations due to untested licenses, copyrights and patents. Due to the high risks and irrevocability, open sourcing proprietary software should not be considered readily. Instead it should be treated as a dangerous maneuver, only used with no other options, and even then only after comprehensive planning.

Successful open source project is not as much a technical phenomenon, as a social one. Nurturing a community requires in depth understanding of users and customers, along with considerable skill in social interactions and facilitating communication.

There are special requirements for successful project to emerge and profitable business to be built on it. Restrictions relate to target industry, used technologies, reputation of the company, architecture of the software, requirements of user interface and other product attributes. Profiting on a successful project requires innovative appropriation vectors and good execution. A successful open source project never guarantees a successful company.

Future studies

Open source is an old phenomenon in an industry that develops faster than imaginable. Many current articles and studies of open source are based on old success stories, but the march of open source is still taking place. Many sources used in this study are old in the scope of software industry. Studies of more recent events would complement this study.

This study has been divided into four different key domains: social, legal, technological and business. For the sake of brevity and scope, these domains have been introduced in a relatively superficial level. These could be expanded on in order to provide an effective “how to”-guide of open source.

6 References

1. Aoki, A., Hayashi, K., & et al. (2001). A case study of the evolution of Jun: An object-oriented open source 3D multimedia library. *Proceedings of International Conference on Software Engineering (ICSE2001), Toronto, CA,,* 524.

2. Bonaccorsi, A., Giannangeli, S., & Rossi, C. (2006). Entry strategies under dominant standards. hybrid business models in the open source software industry. *Management Science, 1*(51), 1085-1098.

3. Bonaccorsi, A., & Rossi, C. (2003). Why open source software can succeed . *Research Policy, 32*(7), 1243–1258.

4. Bonaccorsi, A., & Rossi, C. (2006). Comparing motivations of individual programmers and firms to take part in the open Source movement. from community to business. *Knowledge, Technology and Policy, 18*(4), 40-64.

5. Brooks, F. (1995). Mythical man month. *Paperback,*

6. Bryar, J. (2000). How much does free cost? *The Andover News Network, March 15*

7. Church, J., & Gandal, N. (1992). Network effects, software provision, and standardization. *The Journal of Industrial Economics, 40*(1), 85-103.

8. Curtis, B., Krasner, H., & Iscoe, N. (1988). A field study of the software design process for large systems. *Communications of the ACM, 31*(11), 1268-1287.

9. Dahlander, L., & Magnusson, M. (2008). How do firms make use of open source communities? *Long Range Planning, 41*(6), 629–649.

10. Dahlandera, L., & Magnusson, M. (2005). Relationships between open source software companies and communities: Observations from Nordic firms. *Research Policy, 34*(4), 481–493.

11. DiBona, C., Ockman, S., & Stone, M. (1999). Open sources: Voices from the open source revolution. *Paperback,*

12. Dinh-Trong, T., & Bieman, J. (2004). Open source software development: A case study of FreeBSD. *Software Metrics, 2004. Proceedings. 10th International Symposium,* , 96.

13. Dinh-Trong, T., & Bieman, J. (2005). The FreeBSD project: A replication case study of open source development. *IEEE Transactions on Software Engineering, 31*(6), 481-494.

14. Feller, J., Fitzgerald, B., Hissam, S., & Lakhani, K. (2007). Perspectives on free and open source software. *Paperback,*

15. Fitzgerald, B. (2006). The transformation of open source software.*30*(3), 587-598.

16. Forte, A., & Bruckman, A. (2008). Why do people write for Wikipedia? Incentives to contribute to open-content publishing. *Proceedings of the Proceedings of the 41st Annual Hawaii International Conference on System Sciences,* , 6-9.

17. Gallivan, M. (2001). Striking a balance between trust and control in a virtual organization: A content analysis of open source software case studies. *Information Systems Journal, 11*(4), 277–304.

18. German, D. (2003). The GNOME project: A case study of open source, global software development. *Software Process: Improvement and Practice, 8*(4), 201–215.

19. Ghosh, R., Krieger, B., Glott, R., & Robles, G. (2002). Free/Libre and open source software: Survey and study. *Report to the European Comission, Published Online,*

20. Ghosh, S., Klein, A., Avants, B., & Millman, J. (2012). Innovation by user communities: Learning from open source. *Frontiers in Computational Neuroscience, 6*(12)

21. Godfrey, M., & Tu, Q. (2000). Evolution in open source software: A case study. *Software Maintenance, 2000. Proceedings. International Conference on,* (131 - 142)

22. Gurbani, V., Garvert, A., & Herbsleb, J. (2006). A case study of open source tools and practices in a commercial setting. *Proceedings of the 28th International Conference on Software Engineering,* , 472-481.

23. Hars, A., & Ou, S. (2001). Working for free? motivations for participating in open source projects. *International Journal of Electronic Commerce, 6*(3), 25-39.

24. Healy, K., & Schussman, A. (2003). The ecology of open source software development. *Online Publication,*

25. Hecker, F. (1999). Setting up shop: The business of open source software. *Software, IEEE, 16*(1), 45 - 51.

26. Howe, J. (2006). The rise of crowdsourcing. *Wired Magazine, 14*(6), 1-4.

27. Huia, W., Yoob, B., & Tamc, K. Y. (2008). Economics of shareware: How do uncertainty and piracy affect shareware quality and brand premium? *Decision Support Systems, 44*(3), 580–594.

28. Kenwood, C. (2001). A business case study of open source software. *MITRE Report,*

29. Koenig, J. (2006). Seven open source business strategies for competitive advantage. *IT Manager's Journal, 14*

30. Kogut, B., & Metiu, A. (2001). Open source software development and distributed innovation. *Oxford Review Economic Policy, 17*(2), 248-264.

31. Koski, H. (2005). OSS production and licensing strategies of the software firms. *Review of Economic Research on Copyright Issues, 2*(2), 111-123.

32. Krishnamurthy, S. (2003). An analysis of open source business models. *J. Feller, Et Al. (Eds.): Perspectives on Free and Open Source Software,* , 279.

33. Kuznetsov, S. (2006). Motivations of contributors to wikipedia. *ACM SIGCAS Computers and Society, 36*(2)

34. Lakhani, K., & von Hippel, E. (2003). How open source software works: "free" user-to-user assistance. *Research Policy, 32*(6), 923–943.

35. Lakhani, K., & Wolf, R. (2005). Why hackers do what they do: Understanding motivation and effort in Free/Open source software projects . *Perspectives on Free and Open Source Software, MIT Press,*

36. Langham, M. (2009). The business of open source. *Published Online,*

37. Langlois, R., & Garzarelli, G. (2008). Of hackers and hairdressers: Modularity and the organizational economics of open source collaboration . *Industry and Innovation 15, 2*, 125.

38. Lerner, J., & Tirole, J. (2002). The scope of open source licensing. *Journal of Law, Economics, and Organization, 21*(1)

39. Lerner, J., & Tirole, J. (2005). The economics of technology sharing: Open source and beyond . *Journal of Economic Perspectives, 19*(2), 99.

40. Lerner, J., & Triole, J. (2002). Some simple economics of open source. *Journal of Industrial Economics, 52*, 197-234.

41. Liebowitz, S. J., & Maargolis, S. (1994). Network externality: An uncommon tragedy. . *Journal of Economic Perspectives, 8*(2), 133-150.

42. Lindenberg, S. (2001). Intrinsic motivation in a new light. *Kyklos, 54*(2-3), 317–342.

43. Linus, D. (2004). Appropriating the commons: Firms in open source software. *Chalmers University of Technology,* , 1-23.

44. MacCormack, A., Rusnak, J., & Baldwin, C. (2006). Exploring the structure of complex software designs: An empirical study of open source and proprietary code. *Management Science, 52*(7), 1015-1030.

45. Mockus, A., & Fielding, R. (2000). A case study of open source software development: The apache server. *Software Engineering, 2000. Proceedings of the 2000 International Conference on,* , 263 - 272.

46. Mockus, A., Fielding, R., & Herbsleb, J. (2002). Two case studies of open source software development: Apache and mozilla. *ACM Transactions on Software Engineering and Methodology, 11*(3), 309.

47. Mustonen, M. (2003). Copyleft—the economics of linux and other open source software. *Information Economics and Policy, 15*(1), 99–121.

48. Onetti, A., & Capobianco, F. (2005). Open source and business model innovation. the funambol case. *Proceedings of the First International Conference on Open Source Systems, Genova,*

49. Osterloh, M., & Rota, S. (2007). Open source software development – just another case of collective invention? *Research Policy, 36*(2), 157–171.

50. Rigby, P., German, D., & Storey, M. (2008). Open source software peer review practices: A case study of the apache server. *Proceedings of the 30th International Conference on Software Engineering,* , 541-550.

51. Roberts, J., Hann Il-Horn, & Slaughter Sandra. (2006). Understanding the motivations, participation, and performance of open source software developers: A longitudinal study of the apache projects. *Management Science, 52*(7), 984.

52. Schmidt, D., & Porter, A. (2001). Leveraging open source communities to improve the quality & performance of open source software. *Proceedings of the 1st Workshop on Open Source Software Engineering. Toronto, Canda: ICSE,*

53. Sharma, S., Sugumaran, V., & Rajagopalan, B. (2002). A framework for creating hybrid-open source software communities. *Information Systems Journal, 12*(1), 7-25.

54. Välimäki, M. (2003). Dual licensing in open source software industry. *Systemes d´Information Et Management, 8*(1), 63-75.

55. Välimäki, M. (2005). The rise of open source licensing: A challenge to the use of intellectual property in software industry. *Paperback,*

56. Valloppillil, V. (1998). Open rouce software: A new development methodology. *Published Online,*

57. Viegas, F., Wattenberg, M., Kriss, J., & van Ham, F. (2007). Talk before you type: Coordination in wikipedia. *HICSS '07 Proceedings of the 40th Annual Hawaii International Conference on System Sciences,* , 78.

58. von Hippel, E. (1988). The sources of innovation. *Paperback,*

59. von Krogha, G., Spaetha, S., & Lakhanib, K. (2003). Community, joining, and specialization in open source software innovation: A case study. *Research Policy, 32*(7), 1217–1241.

60. Weber, S. (2005). The success of open source. *Paperback,*

61. West, J. (2003). How open is open enough?: Melding proprietary and open source platform strategies. *Research Policy, 32*(7), 1259–1285.

62. West, J., & Gallagher, S. (2004). Key challenges of open innovation: Lessons from open source software. *Published Online,*

63. West, J., & Gallagher, S. (2006). Challenges of open innovation: The paradox of firm investment in open source software. *R&D Management Volume, 36*(3), 319–331.

64. West, J., & O'Mahony, S. (2005). Contrasting community building in sponsored and community founded open source projects. *Proceedings of the Proceedings of the 38th Annual Hawaii International Conference on System Sciences, 07*

65. Yilmaz, C., Memon, A., Porter, A., Krishna, A., Schmidt, D., & Gokhale, A. (2006). Techniques and processes for improving the quality and performance of open source software. *Software Process: Improvement and Practice, 11*(2), 163–176.