Master thesis

Innovation dynamics in open source software

Author:

Name:	Remco Bloemen
Student number:	0109150
Email:	remco.bloemen@gmail.com
Telephone:	+316 11 88 66 71

Supervisors and advisors:

Name:	prof. dr. Stefan Kuhlmann
Email:	s.kuhlmann@utwente.nl
Telephone:	+31 53 489 3353
Office:	Ravelijn RA 4410 (STEPS)

Name:	dr. Chintan Amrit
Email:	c.amrit@utwente.nl
Telephone:	+31 53 489 4064
Office:	Ravelijn RA 3410 (IEBIS)

Name:	dr. Gonzalo Ordóñez–Matamoros
Email:	h.g.ordonezmatamoros@utwente.nl
Telephone:	+31 53 489 3348
Office:	Ravelijn RA 4333 (STEPS)

Abstract

Open source software development is a major driver of software innovation, yet it has thus far received little attention from innovation research. One of the reasons is that conventional methods such as survey based studies or patent co-citation analysis do not work in the open source communities. In this thesis it will be shown that open source development is very accessible to study, due to its open nature, but it requires special tools. In particular, this thesis introduces the method of dependency graph analysis to study open source software development on the grandest scale. A proof of concept application of this method is done and has delivered many significant and interesting results.

Contents

List of Figures

List of Tables

Thesis outline

Chapter one will quickly introduce open source software, what it is, how it works and why it is interesting to study its innovation dynamics. It particularly looks at the intellectual property debate with respect to software patents, which is the original motivation for this thesis. It will be identified that a major problem in this debate is the lack of methods to analyse innovation dynamics in the open source software world. The rest of this thesis will be focused on developing a method to analyse innovation dynamics in the open source world.

In short the method will analyse how the interdependencies among open source projects develop over time. Like any other technology, software can be seen as build from parts that are combined to create new parts. The open source projects are effectively all developing a particular part. In doing so they rely on other projects developing their parts. For example a music player project can rely on a music reader project and on a audio driver project (in reality there are many more parts involved). These projects and their interdependencies form a directed graph which changes over time. By analysing this graph and its changes one can gather information on the underlying innovation.

Section 1.4 will explain the need for such a method in the software patent debate. Currently the U.S. patent office has a quarter million patents that make claims related to software development. Although the U.S. law prohibits patenting inventions without physical existence, which software arguably is, court rulings have extended this to include "anything [...] made by man". The situation in Europe is different, the European Patent Convention explicitly forbids software patents, but the pressure to conform to the U.S. system is large. As a consequence, studies have been done to investigate innovation in the software sector and the effect software patents would have. As chapter 1.4 will argue, the methods these studies employ are basically blind for non-commercial software development, therefore a new methods are required.

Chapter 1

Open source software

The open source software community offers a very interesting and mostly unexplored opportunity to research innovation. The open source software community has shown itself to be an important driver of innovation in the IT industry, with some crucial pieces of IT technology developed by open source projects and a software developer workforce that outnumbers the entire U.S. commercial software developer workforce. The open source software model has inspired similar models in, among others, art, hardware development, biotechnology. A funny example is the open cola project, designed to explain the concept of open source and mocks Coca Cola's use of trade secrecy by developing a cola completely in the open. In table 1.1 one can find the current recipe, if someone improves the recipe, he is required to share his discovery as well.[1]

The open source model has an interesting interaction with commercial development. Many large and small commercial entities are using and/or investing in open source development. But there are also conflicts, for example when commercial entities break the license agreements of the open source projects, or when open source projects break patents held by commercial entities. Currently there is an interesting and important debate going on about the nature and value of software patents, which has wide consequence for both commercial and open source development. It is important to provide this debate with the scientific evidence required to come to an optimal, fair and rational solution.

Due to the nature of open source a lot of information can be gathered in an automated fashion with relatively little effort, yet this area is still very unexplored. In open source development large projects are taken on by individuals who can live in opposite sides of the world, usually their only means of coordination is through the internet. The development can happen through various channels, a common pattern is to have a website to supply users and new contributors with information, a mailing list and to discuss the development process, an issue tracking systems to administrate what needs to be done and a revision management system to track what has been done in the past. And here is the good part: in open source, all these systems are publicly accessible, allowing innovation researchers to have almost perfect information about the innovative process through the entire history of the project.

In this chapter a short introduction is made to the structure of the open

[1] Interestingly, the project started with a recipe found in the diary of the inventor of Coca-Cola, which has since fallen in the public domain.

Table 1.1: The recipe for OpenCola, version 1.1.3.

Flavouring:	
3.50 ml	orange oil
1.00 ml	lemon oil
1.00 ml	nutmeg oil
1.25 ml	cassia oil
0.25 ml	coriander oil
0.25 ml	neroli oil
2.75 ml	lime oil
0.25 ml	lavender oil
10.0 g	gum arabic
3.00 ml	water
Syrup:	
10.0 ml	flavouring formula
17.5 ml	phosphoric acid
2.28 l	water
2.36 kg	plain white sugar
30.0 ml	caramel colour
2.5 ml	caffeine (optional)
Soda:	
1 part	syrup
5 parts	carbonated water

Source: `http://www.colawp.com/colas/400/cola467_recipe.html`
License: GPL Version 2.

Figure 1.1: The entire original BSD license

> Copyright (c) *year copyright holder*. All rights reserved.
>
> Redistribution and use in source and binary forms are permitted provided that the above copyright notice and this paragraph are duplicated in all such forms and that any documentation, advertising materials, and other materials related to such distribution and use acknowledge that the software was developed by the *organization*. The name of the *organization* may not be used to endorse or promote products derived from this software without specific prior written permission.
>
> THIS SOFTWARE IS PROVIDED "AS IS" AND WITHOUT ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Source: Wikipedia, originally from the Regents of the University of California.
License: Public domain

source community, some of the major organisations and projects are introduced, their development and cooperation methods are explained and the concepts of packages and dependencies are explained. For a more elaborate introduction to open source from an innovation perspective the reader is kindly referred to (St.Amant and Still, 2007) and in particular (Deek and McHugh, 2008).

1.1 The open source licenses

There is no single open source philosophy that all developers subscribe to. Some are of the opinion that all software development should be open and that use of commercial software should be actively discouraged. Others take a more relaxed stance and want their software to benefit others in any way it can, whether it is commercial or not. It is impossible to catagorise all the different opinions, but it is possible to study their brainchilds, the opensource software licenses.

Opens source software licenses fall broadly in three categories depending on how much commercial use is prohibited by the license. First there are the permissive licenses, such as the MIT License and the BSD Licenses. These licenses pose very little constrains on the use of the source code, they are quite comparable with releasing the source code in the public domain. The license usually has a disclaimer, "the author takes has responsibility whatsoever", and sometimes contain an attribution term, "the original authors must be credited in derivative works". Some less serious variations state that the user can "do what the fuck [he] want[s] to" (Hovecar, 2004) or have a clause stating that the user is "encouraged to buy the author a beer" (Kamp, 2004). These licenses tend to be very short, the original BSD license is printed in it entirety in figure 1.1. Opposite of these licenses are the strong copyleft licenses, such as the popular GPL license. These licenses have a reciprocal nature, any derivative works must also be released under the same license terms. Where the permissive licenses allow the work to be integrated in a commercial software package, the strong copyleft license require this software package to be released in a strong copyleft license as well. The license is designed to prohibit use in commercial software. Besides the reciprocal clause and attribution clause some other popular clauses

are a patent retaliation clause, which revokes the license as soon as the user use patents in a way that may harm the project and a DRM restriction clause that revokes the license when the final product limits the end user in freely using the product by other means than modifying the software (such as modifying the hardware).

For some developers the permissive licenses are to free, because it allows other authors to use a piece of technology without ever contributing their improvements back to the original author and the strong copyleft licenses are to strong, because it prevents users of the technology from releasing their composite product under different license terms. The weak copyleft licenses are a compromise, the user is allowed to use the technology as a component in a larger product which is released under a different license, but any changes made to the component must be released under the same weak copyleft license. An additional clause stipulates that even though the source code of the composite projects does not have to be released, provisions mus be made so that user can see, modify or replace the weak copyleft component. Effectively only the part that was open source in the first place must become open source. Such license are popular with commercial developers, the WebKit engine, which is used by both Apple and Google as the core of their web browsers, is under the LGPL license. This license allows them to develop their own proprietary web browsers, but also requires them to share improvements in the WebKit engine with the world (and thus each other).

It is important to note that handing out the source code under a certain license does not change the fact that the code author still owns the copyright. Possession of the copyright allows the owner to re-release the code under different licenses. In the dual-license businesses model a company releases a product in two versions, one in an open source license and one in a commercial license. If the open source license is of the strong copyleft variety then commercial users are required to pay for a commercial license. But even if the open source license is permissive, the commercial version may be interesting due to proprietary extensions or commercial support.

Project that are organised in a nonprofit or for profit organisation usually want to retain all the copyright in this central organisation. This is required when the organisation needs to change the license terms of the code, or for the dual-license scheme. To maintain the copyright over all the code the organisation is required to obtain copyright waivers from all the developers all over the world. This is a very complex a juridical manoeuvre that very few software developers really care about.

1.2 Commercial involvement in open source

Open source software development is not the opposite of commercial software development. Successful products and businesses have been set up around open source packages to provide commercial support. Also, companies have released commercial products in the open source to further the development of the project. Three quite famous cases are Firefox, Chrome and LibreOffice.

Firefox In 1994 Netscape pioneered the web browsers market with its commercial Netscape Navigator product. In March 1998 Netscape released most of

the browsers source code as an open source project called the 'Mozilla Application Suite'. This in turn formed the basis for what is now the popular open source web browser Firefox.

Chrome In 1998 the KDE project started implementing their own open source browser engine called KHTML based on earlier work. In 2001 Apple forked the KHTML code (Controversially, they announced this to the KHTML developers only after they worked on the fork for a year. This contributed to a divergence between the two projects that made sharing improvements difficult. Apples difficulty with sharing its improvements with the KHTML developers let to some bad publicity. Eventually the situation improved and now both projects coexist and collaborate.) Apple's forked KHTML engine was developed into the WebKit browser engine, which inherited KHTML's open source license. This WebKit engine drives Apple's closed-source Safari web browser used on Mac OS X and the iPhone Operating System. Google used WebKit as the engine for its Chrome browser, which in turn was largely released under an open source license.

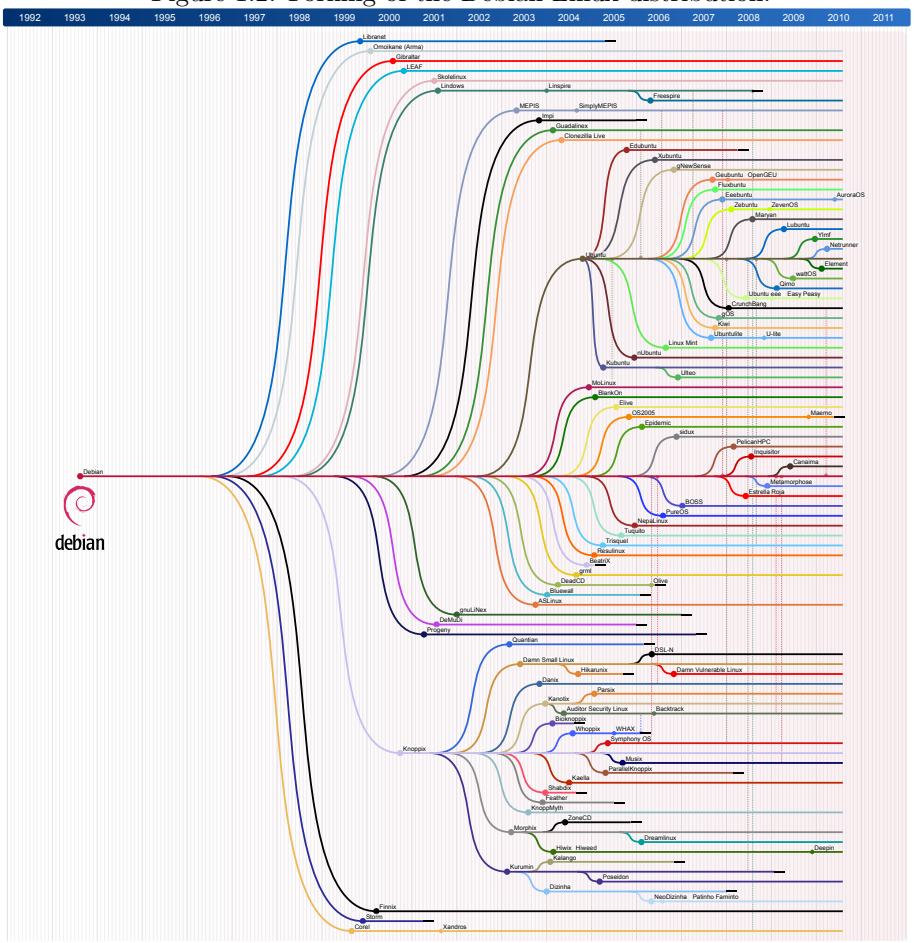
LibreOffice Sun Microsystems acquired StarOffice in 1999, continued to develop it and in 2000 released the source code under and open source license. The open source fork became known as OpenOffice. Sun continued to sell StarOffice as a version of OpenOffice with proprietary extensions. Sun continued to invest development resources in OpenOffice until Sun itself was acquired by Oracle Corporation. Developers feared Oracle might discontinue the investment in OpenOffice or otherwise harm the project so many developers forked the project into the LibreOffice open source project. When Oracle did discontinue all OpenOffice involvement in 2011 Google and five other organisations stepped up and each devoted one employee to the project.

1.3 Opens source development

The development process in open source software project is very dynamic. On a given project, developers come and go. Some developers stay around for years and contribute major parts, other times a developer contributes a single bug fix and is never heard from again. Often, developers can come from all over the world and the group is diverse, but sometime a project may have a majority of their developers originating from a single company. In any case a means of coordination is required that can cope with a dynamic pool of developers that are not geographically close. Therefore almost all the development and related processes happen online using various tools such as mailing lists, wiki's, issue trackers, revision managers, etcetera. Usually all these systems are publicly accessible, in spirit with the open source philosophy and to facilitate the self-education of new developers.

Forking Juridically every slight change made to a software package a makes it a new work which is derivative of the original work. The copyleft licenses require this work to be released under the same terms as well. This would mean that every small change would mean an entirely different code base, which can in turn be used to create many different derivative works from. In practise, the

Figure 1.2: Forking of the Debian Linux distribution.

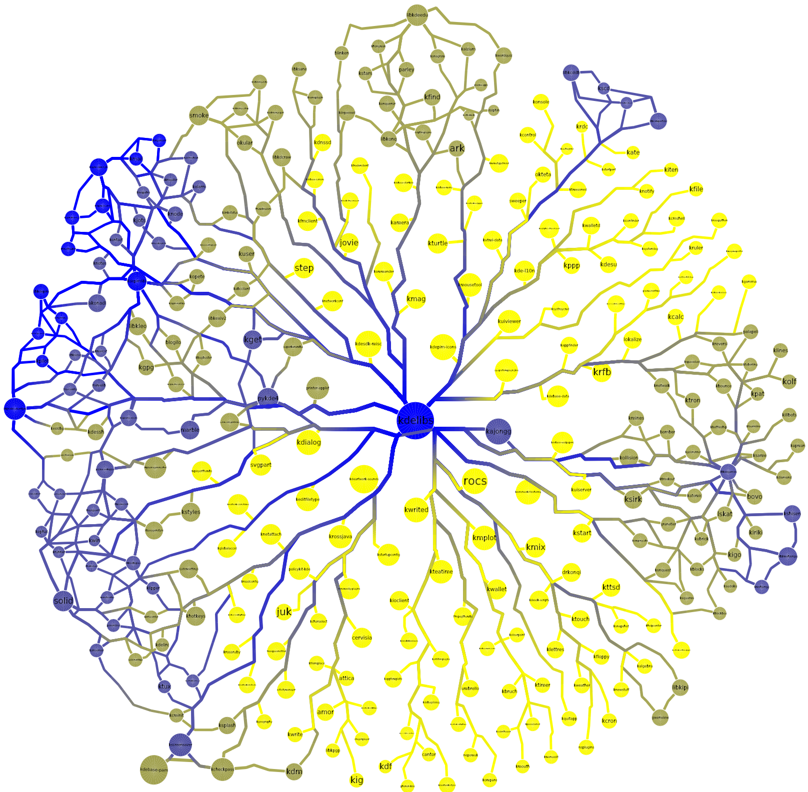


Source: `http://futurist.se/gldt/` (sligthly modified)
License: GNU Free Documentation License.

community maintaining a particular software package will integrate all these changes back into a single normative code base. The community will then periodically release new versions of this code base.

This process of converging to a single version requires the developers to agree on the direction to go in. Often they succeed in this, but sometimes a derivative work does not integrate back in to the mainline and becomes an open source project on its own. This process is called 'forking'. There can be many reasons for a fork, such as specialisation in different direction, disagreement about license terms or copyright ownership or an experimental design decision. A fork does not necessarily imply a failure on the part of the forking or the original developers to keep coherence. In figure 1.2 one can see an example of forking behaviour. Debian is one of the first general purpose Linux based operating systems and over a period of twenty years many projects have taken Debian as a basis to create more specialised operating systems.

Figure 1.3: Internal dependencies of modules in the KDE community. Colour represents the k-core measure. The graph edges have been bundled to improve readability. Source: own illustration, created using Tulip.



Dependencies The open source community consists of numerous projects that produce periodical releases, called *packages*. Usually these projects rely on technology from many other projects to function. Consequently, the packages require the packages from the other projects to be installed as well. When this is the case required package is called an *dependency* of the former package and conversely, the former package is a *dependee* of the required package.

These packages and their dependency relations can be considered graphs. In figure 1.3 the dependency relations of all the projects of the KDE community are shown. Only dependencies within the KDE community are shown, external dependencies are left out. The colouring represents the k-core measure, which reveals clustering of modules. The graph shows All modules depend on kdelibs (centre) and some form clusters around specific technologies such as the games, the email and address book applications.

1.4 The intellectual property debates

Currently there are heavy debates on the subject of intellectual property (IP). With the advent of computers and the internet the economic cost of information

reproduction and distribution has become negligible. The result is that there is a vast online community that exchanges information freely, a lot of which are products of own work released to the public domain and some are illegal reproductions of material covered by IP protection. The ease of copying and sharing is at odds with the current intellectual property legislature, which was mostly written in a time when copying books and films required printing-presses and film development equipment. This has led economists and other authors, such as Boldrin and Levine (2008) and Boyle (2008) to the conclusion that the intellectual property protection needs to be heavily reformed, or even abolished. They consider the success of creators in industries that are not covered by IP protection legislature and those that deliberately do not use IP protection as proof that IP incentives are not necessary to stimulate creation. Needless to say, many large patent and copyright holders disagree.

The debate is particularly fierce in the media industry, where IP protection is largely done using copyrights and in the software industry, where both copyright and patenting provides IP protection. Both areas have stakeholders that range form hobbyist public domain producers to large global corporations and both deal with IP infringement on a massive scale. Although the media industry debate is interesting on its own (see Boldrin and Levine, 2008; Boyle, 2008, and many others), this thesis will focus on the software industry. Particular emphasis will be given to software patents since the US and EU has a painful conflict in this area that many would like to see resolved, but few can agree on how.

1.4.1 The software patent debate

Goldstein (2005) explains that patents provide the holder of the patent with a temporary monopoly on an invention. This monopoly allows the holder to exploit his invention as he wishes without having to worry about competitors stealing his idea. In order to obtain a patent an invention has to satisfy four demands. First, the subject of the invention has to be *patentable*, for example in all legislatures the invention of an industrial machine will be patentable but the invention of a story line for a book will not be patentable. The second demand is *utility*, the invention must solve the problem it is designed to address, that is, the invention must work. According to (Jaffe and Lerner, 2007, page 28) this requirement is not important in practise, since almost anything can be shown potentially useful in some way. Thirdly, the invention must be *new*. If someone can proof that he knew about the invention before the patent application than this is called prior-art and the invention will not satisfy the novelty requirement. The fourth requirement builds upon this, the invention has to be *non-obvious* to a person skilled in the art at the time of invention. This demand prevents someone from acquiring patents on slight variants of already known inventions.

The last three demands vary only in details between nations, but the first demand, which subject are patentable and which not differs widely regarding software. In the US patent law, patents can only be awarded for inventions which have some physical existence or processes resulting in physical products. However, various court cases have stretched this law to include business methods, financial constructions and computer software. This culminated in 1980 when the Supreme Court judged in the Chakrabarty decision that “anything under the sun made by man” is patentable Jaffe and Lerner (2007). Even though the

Table 1.2: U.S. patents awarded for software inventions up to and including 2009.

Class	class title	patents
	Data Processing:	
700	Generic Control Systems or Specific Applications	15,747
701	Vehicles, Navigation, and Relative Location	17,197
702	Measuring, Calibrating, or Testing	17,050
703	Structural Design, Modeling, Simulation, and Emulation	5,317
704	Speech Signal Processing, Linguistics, Language	10,015
705	Financial, Business Practice, Management, or Cost/Price Determination	12,231
706	Artificial Intelligence	3,981
707	Database and File Management or Data Structures	19,690
715	Presentation Processing of Document, Operator Interface Processing, and Screen Saver Display Processing	11,848
716	Design and Analysis of Circuit or Semiconductor Mask	8,071
717	Software Development, Installation, and Management	6,803
	Electrical Computers and Digital Processing Systems:	
708	Arithmetic Processing and Calculating	7,800
709	Multicomputer Data Transferring	21,959
710	Input/Output	16,061
711	Memory	19,083
712	Processing	7,855
713	Support	14,157
718	Virtual Machine Task or Process Management or Task Management/Control	2,506
719	Interprogram Communication or Interprocess Communication (Ipc)	2,598
	Other:	
714	Error Detection/Correction and Fault Detection/Recovery	22,780
720	Dynamic Optical Information Storage or Retrieval	3,034
725	Interactive Video Distribution Systems	3,860
726	Information Security	4,490
Total software patents:		254,133
Total patents:		4,015,989

Source: data from PTM (2010)

Table 1.3: Breakdown of respondents by activity and whether they release their software in mostly as open source.

	Verkade et al.	Blind et al.	mostly open source
Independent developers	0	38	82%
Software bureaus	4	139	8%
Other busineses	2	58	8%
Non-commercial	1	0	
Patent experts	7	0	

Source: data from Verkade et al. (2000); Blind et al. (2005)

original law still stands, in practise it is rendered irrelevant as the US system now allows the patenting of algorithms and other non-physical inventions. So far, this has resulted in a quarter million patents for software inventions, 6% of the total number of patents, see table 1.2.

In Europe the situation is a bit different, the European Patent Convention (EPC) explicitly states that "discoveries, scientific theories and mathematical methods" and "schemes, rules and methods for performing mental acts, playing games or doing business, and programs for computers" are not regarded as inventions (EPO, 2000, article 52). According to Verkade et al. (2000) this legislature has been incorporated in national law, but Verkade et al. (2000) are surprised by the amount the European Patent Office dares to deviate from these laws. Much like the US case the law has been bend in practise till the point where its relevance can be questioned. In 2008 the president of the EPC, Alison Brimelow, officially questioned the European Patent Office with regard to software patents to the Enlarged Board of Appeal of the EPC. In 2010 the board gave a 55 page opinion that concludes that the case law is consistent and the president therefore has no right to question it. On their website the EPO publishes:

> If a claim related to a computer program defines or uses technical means it is not excluded from patentability as a computer program 'as such'. However, only those aspects of a claim which contribute to its technical character are taken into consideration for assessing novelty and inventive step. — epo (2010)

The official explanation of the current case law is in **?**.

1.4.2 The open source blind spot

The software patent debate has been going on for at least a decade and has inspired, amongst others, Verkade et al. (2000) and Blind et al. (2005) to study the potential consequences of software patents. Both research programs where set-up as a literature study and a survey of stakeholders. Verkade et al. (2000) did a study on the development of the relevant Dutch and European laws and a survey among commercial software developers, non-commercial software researchers (universities) and patent offices. Blind et al. (2005) start with a literature study and proceed with an extensive survey among commercial software developers. They surveyed independent software developers, software bureaus and other businesses requiring software development (electrotechnology,

Table 1.4: Literature search on social network analysis (SNA) and software development (SD) in innovation journals.

Journal	SNA	SD	SNA and SD	filtered
Research Policy	239	162	26	4
Technovation	116	92	7	0
Tech. Forecasting and Social Change	142	89	5	2
Scientometric	233	23	4	2
Total:	730	366	42	8

Source: own creation, searched using ScienceDirect and Google Scolar.

telecommunication, etc.). The study of Blind et al. (2005) is particularly interesting because they asked how often the developers release their code in the public domain. Table 1.3 provides a breakdown of the respondents to both studies.

The studies may have overlooked the non-registered developers. Blind et al. (2005) developed their sample list in co-operation with the German Federal Ministry of Economics and Technology (BMWi) and they selected their addresses by the economic class they where registered under. The BMWi also supplied them with a list of independent developers. Although Blind et al. (2005) do not mention this, it appears that these are all software developers registered at the trade office. The study of Verkade et al. (2000) contains a list of respondents, none of them are independent developers. It is therefore likely that non-registered developers are entirely overlooked.

A large fraction of the open source developers are however not registered as independent developers at trade offices. Many open source developers work on open source projects in their spare time and have jobs as commercial developers or academics.

So, even though the study by Blind et al. (2005) includes some open source developers, the large majority of open source developers is overlooked. One could even argue that the included open source developers are biased towards the commercial side, since they have registered themselves as commercial developers at the trade office.

Another area where open source software (and perhaps to some extend software in general) is overlooked is in patent citation analysis. As will be explained in more detail later, this method of analysing innovation dynamics involves the use of large patent databases where one looks at how patents cite each other. The resulting networks can be analysed using social network analysis techniques. These networks are then used to gain insights in the evolution of a certain class of innovations. This method is of course blind for open source software, since their licences prohibit the use of patents. Furthermore, the complicated position of software patents may invalidate the use of the patent citation analysis technique for software in general.

Table 1.5: The papers resulting from the literature search. The nature of their networks—the nodes, relations and dataset—is shown.

Article	Nodes	Relations	Dataset
Research Policy:			
Dahlander and Wallin (2006)	Mailing list posters	Replies	Gnome-dev mailing list
Dittrich et al. (2007)	Software companies	Strategic alliances	MERIT-CATI, CGCP
Engelsman and van Raan (1994)	Patents classes	Word co-occurence, co-classification	EPAT, WPI/L
M'Chirgui (2009)	Smart-card firms	R&D alliances	SCIFA
Scientometic:			
McCain et al. (2005)	Authors	Co-citation	Journal of Software Engineering
Lim and Park (2010)	Patent classes	Word co-occurence, co-classification	WIPS
Technological Forecasting and Social Change:			
He and Hosein Fallah (2009)	Patents	Investor–assignee	USPTO BIB database
Choi et al. (2007)	Patents classes	Cross impact	USPTO filtered for the ICT industry

Source: own creation.

1.5 Litterature search on network analysis in software development

To find existing literature on the analysis of software innovation dynamics using network analysis a literature search was executed. Four important journals for innovation research where searched: Research Policy, Technovation, International Journal of Technological Forecasting and Social Change and Scientometric. The first three were searched using ScienceDirect, the last using Google Scolar. The journals were search for two concepts, network analysis and software development, with the search query `"network analysis" OR "social network"` and `"software development" OR "software innovation"` respectively. Several hundreds of papers where found, as is displayed in table 1.4. The two queries were then combined which resulted in a more manageable 42 hits. These articles where then scanned by hand to filter out the false positives, 33 articles were eliminated because they did not employ network analysis and one article was eliminated because it had no relation to software development.

The eight remaining articles where then studied in more detail to analyse the nature of the networks they analyse. The nature of the nodes, the relations and the origin of the data is presented in table 1.5.

Half of the articles used networks based on patents, but these papers look at patents on the grandest scale where software development is only a small part of the whole. No studies where found that used patent network analysis to specifically investigate software innovation, but this is hardly a surprise given the controversial nature of software patents.

Two of the four remaining studies used companies and their alliances as the network. M'Chirgui (2009) analysed the strategic alliance network of IBM to investigate IBM's transformation from a hardware manufacturer to a software service provider. Using these networks the authors show how very large companies can quickly change their strategy by consciously changing their alliance network. M'Chirgui (2009) analysed the R&D alliance network of smart-card firms to demonstrate that there is a strong correlation between these networks and the direction in which the technology develops. Both these studies provide

interesting conclusions that might be applicable to the open source community, if one substitutes "project" for "company". But since they use proprietary databases that contain only companies they are inherently blind for open source software development, as explained in section 1.4.

Chapter 2

Theoretical background

This research will draw on two major theoretical frameworks, the theory of innovation dynamics and the theory of social network analysis. These two frameworks will be combined to develop a framework wherein the research question and hypotheses can be formulated as empirically testable statements. The theory of innovation dynamics concerns the processes and outcomes of problem solving in organisations. In particular it concerns research and development, inventions and their adoption by others. The theory of social network analysis concerns individuals or organisations which have certain relations with each other. The relations can be anything from friendship and kinship to email communications to business transactions and patent co-citations. The resulting networks can be analysed using general techniques to gain insights in the processes and structures that produce them. Combining these two theories is not new however, it has been done before in patent co-citation analysis. A literature study is therefore employed to find existing uses of social network analysis on software development in the innovation research journals.

2.1 Theory of innovation dynamics

To do an explorative empirical study in the innovation dynamics a clear definition of the relevant concepts and theories is required. There is little existing innovation dynamics research in the open source software community to build on, so it will be necessary to derive a usable theoretical framework. Luckily, there is a large existing body of research on the innovation dynamics in other areas, for example in consumer technology and agriculture, which has resulted in a clear concepts and theories. In this section the most relevant results will be introduced.

2.1.1 What is innovation?

According to (Narayanan, 2001, page 67) innovation is commonly held as being synonymous with invention, referring to a creative process whereby two or more existing entities or ideas are combined in a some new way to produce a configuration not previously known by the firm or person involved. The Oslo Manual, a well respected body of guidelines in the field of innovation research,

presents the following definition:

> An *innovation* is the implementation of a new or significantly improved product (good or service), or process, a new marketing method, or a new organisational method in business practises, workplace organisation or external relations.
>
> — Definition from the The Oslo Manual (2005)

Josep Schumpeter, an early economist interested in development, was the first to distinguish between invention and innovation. Schumpeter considered an invention to be a new combination of preexisting knowledge whereas an innovation is broader. If an entity produces a good or services or uses a system or procedure that is new to it, it makes an innovation. An invention is thus always part of an innovation, but not all innovations need to involve inventions. (see Schumpeter, 2004, page xix and note 32). In this view innovations include the creation of a technological change new to the company and the use of an existing invention by a company which did not use it yet. An example of the later would be the adoption of bar-code scanners by super markets.

This results in two practical uses of the word innovation, it can either refer to a particular artifact or the process of creating and using an artifact. Narayanan (2001) explicitly uses both interpretations, he uses the terms *innovation process* for the process of arriving at a technical solution to a problem and *innovation output* to the solution itself. In this research paper the same disambiguation will be used where necessary.

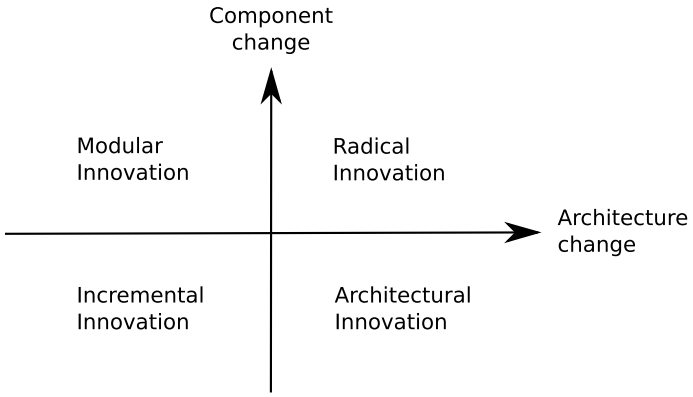
2.1.2 The Henderson-Clark classification

Innovations can be classified by the extend to which they change the existing products or processes. Innovations that leave the existing products or processes relatively unchanged are called incremental innovations, an example is an increase in resolution in computer screens. The other end, where an innovation involves a new approach to an existing product or process is called a radical innovation, an example is the move from cathode ray tube (CRT) to thin-film transistor (TFT) technology in computer screens.

It should now become apparent that a classification of innovations is very much context dependent. The shift from CRT to TFT might be a radical innovation in context of computer screens, they are only an incremental innovation in the context of public addressing systems, such as screens displaying flight departure times in airports. To give another example, in the context of cars slightly increased power would be considered an incremental innovation, but it might entail a radical innovation such as a turbocharger in the context of internal combustion engine technology.

When Henderson and Clark (1990) researched the success factors of innovations they found that an incremental versus radical dichotomy was not enough to classify innovations. They divided innovation along two dimensions, component change and architecture change. The first dimension measures the amount in which specific technologies in an innovation depart from earlier ones. The second dimension measures the amount in which configurations among technologies in an innovation depart from earlier ones. (see also Narayanan, 2001, pages 72–74)

Figure 2.1: The Henderson-Clark classification of innovation.



Source: based on Scocco (2006), created using Inkscape.

Although Henderson and Clark (1990); Narayanan (2001) also focus on the organisational and knowledge aspects of the innovation, this research will mainly focus on the technological aspects of innovations. Much like in Scocco (2006)'s version the model will be narrowed down to the technology and the knowledge management aspects will be left out. It will become evident that the classification remains valuable.

Figure 2.1 shows the classification when the spectrum of innovation is quartered by along the dimensions. The four quadrants represent four classes of innovations, the familiar incremental and radical innovations and the new modular and architectural innovations. According to Narayanan (2001); Scocco (2006) they represent:

Incremental innovations are minor improvements to existing products, technologies or practises. Typical examples would be improvements in the technical specifications of a product. For example, in the context of hard disks a slightly larger capacity is considered an incremental innovation.

Modular innovations are significant changes in elements of existing product, technologies or practises without significant changes in the composition of the elements. For example in the context of cars the replacement of an analog speedometer with a digital speedometer is considered a modular innovation.

Architectural innovations use existing elements but link them in different ways. In the context of ceiling-mounted fans the invention of a portable fan would be an architectural innovation since the components—the fan blade, motor and control system—would be mostly the same but the architecture of the product would be different.

Radical innovations replace the existing architecture and components with something new. Returning to the hard disk example, the introduction of hard disks based on solid state memory is a radical innovation.

Narayanan (2001) mentions three major motivations for the classification. The four classes differ in the process of innovation, they differ in the economic impact and they differ in the role of a manager in the innovation process.

In the open source community there is hardly any economics and manage-

ment, but these motivations have analogous interpretations. The economic impact can be abstracted to the amount the innovation affects its environment, or in the case of OS, how much the innovation is used. The role of the manager in the innovation process is taken by the developers, often there is a single developer that takes charge of a specific idea and implements it. Should the idea be too big for a single developer to implement he would try to gain support from the community while making initial steps. This initiating developer is therefore a natural manager for a specific innovation.

2.1.3 The Bass diffusion model

Once an innovation is released to the public a process starts where an increasing portion of the market decides to use the innovation. In the theory of innovation dynamics this process is called diffusion and the users are called adopters. (see Narayanan, 2001, chapter 4)

To model the process of innovation diffusion Bass (1969) introduces two processes that propagate an innovation. The first processes is involves individuals that decide to use an innovation based on their perception of its merits, without looking at the experiences of others. The second process involves the word-of-mouth effect or the bandwagon effect, individuals adopt the innovation solely because they hear of the experiences of previous adopters. Of course in reality, everyone will be somewhere in between these two extreme types, but for the sake of modelling it suffices to consider the relative abundance of both types.

It should be noted that Bass (1969) and all later authors, use confusing terms to describe the two types of adopters. The first type are called "innovators", not to be confused with those actually inventing the innovation and the second type are called "imitators", not to be confused with those developing imitating offerings. If one remembers that the model concerns the demand side of the market and not the supply side than it will all be clear.

To model the diffusion process, let M be the total market size for the innovation and A the current number of adopters, such that $0 \geq A \geq M$. The two adoption processes can then be described as follows: (see also , BBRI; Vijay Mahajan, 1990)

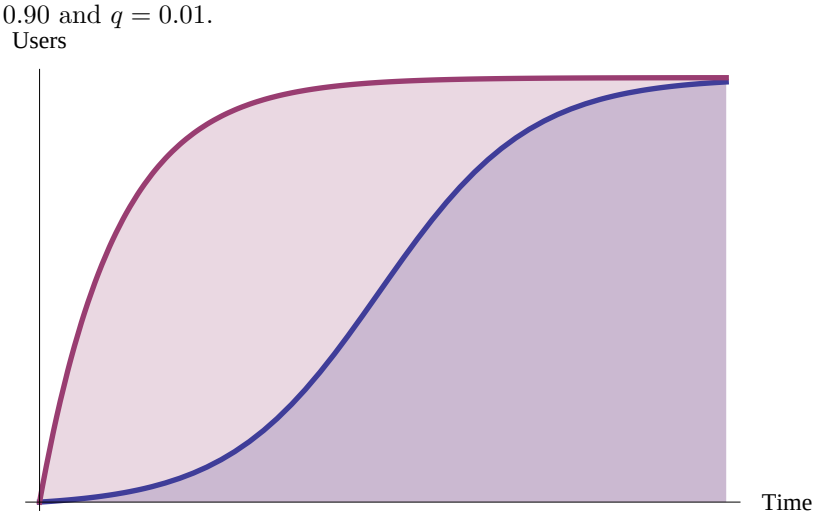
Innovators: Some individuals in the market that don't use the innovation might decide to adopt the innovation. The rate at which this happens is p, the coefficient of innovation. The number of user that do not use the innovation is $M - A$, so the inflow of adopters is $p(M - A)$.

Imitators: The people who use the innovation can express their fondness to people who do not yet use the innovation, which can influence them to adopt the innovation. The rate at which this happens is q, the rate of imitation. The number of user that do not use the innovation is again $M - A$, the chance of meeting someone that does use the innovation is proportional to $\frac{A}{M}$ so the inflow of imitators can be modelled as $q\frac{A}{M}(M - A)$.

When these two effects are combined, the net inflow of users, represented by the time derivative of A, can be modelled as:

$$\begin{aligned}\frac{\mathrm{d}A}{\mathrm{d}t} &= p(M - A) + q\frac{A}{M}(M - A) \\ &= \left(p + q\frac{A}{M}\right)(M - A)\end{aligned}$$

Figure 2.2: The Bass model of innovation diffusion. The blue trace represents a Bass diffusion with $p = 0.01$ and $q = 0.90$. The purple trace bass diffusion with $p = 0.90$ and $q = 0.01$.



Source: own illustration, created using Mathematica.

Often one is not concerned with the market size M and only interested in the fraction of the market—denoted with F—that uses the innovation. Of course, $F = \frac{A}{M}$. By dividing the above equation with M one obtains the Bass model:

$$\frac{\mathrm{d}F}{\mathrm{d}t} = (p + qF)(1 - F) \tag{2.1}$$

Bass (1969) solves this ordinary differential equation, which results in the following function for F:

$$F(t) = \frac{1 - \mathrm{e}^{-(p+q)t}}{1 + \frac{q}{p}\mathrm{e}^{-(p+q)t}} \tag{2.2}$$

In figure 2.2 two Bass diffusions are plotted using equation (2.2), one representing an innovation diffusion with many innovators and few imitators and one representing a diffusion with many imitators and few innovators. When there are more imitators it takes a while for the innovation to take of since the majority of the potential users are waiting for someone else to try it first.

To fit the model of equation (2.2) to empirical data two additions are necessary. First the market size has to be re-introduced, this is done by taking $A(t) = MF(t)$. The market size, M, in this equation represents the total number of potential adopters for this specific product, not the total number of adopters for a category of products. Since it is not possible to know in advance who will eventually be using an innovation it is difficult to determine M in advance. Furthermore, the Bass model assumes the market size to be constant and competition free, which is unlikely in practise. Therefore the quantity has to be fitted to the data, statistical goodness-of-fit measures can then be used to determine the validity of the assumptions. The second addition is the time

Table 2.1: Overview of the variables and parameters of the Bass model as presented in equation (2.3).

	Dimension	description
A	adopters	adopters at a the model time
t	time	model time
t_0	time	time of innovation introduction
M	adopters	number of potential adopters
p	time^{-1}	rate of adopter innovation
q	time^{-1}	rate of adopter imitation

Source: own creation.

at which the innovation is introduced. Until now the assumption was that the introduction was at $t = 0$, in arbitrary units. For empirical data fitting it is necessary to be able to specify an arbitrary introduction time. This can be achieved by introducing the introduction time t_0 in the equation as $A(t) = MF(t - t_0)$. Again, this variable can be fitted if it can not be determined in advance.

$$A(t) = M\frac{1 - \mathrm{e}^{-(p+q)(t-t_0)}}{1 + \frac{q}{p}\mathrm{e}^{-(p+q)(t-t_0)}} \tag{2.3}$$

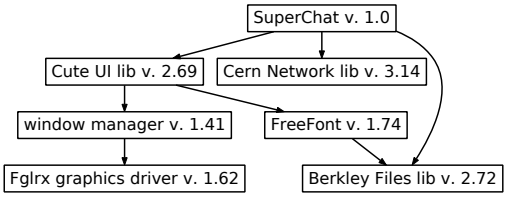
Equation (2.3) incorporates the two additions and can be readily applied to empirical data. In Vijay Mahajan (1995) and many other empirical studies this happens in the differential form, since only absolute sales figures are available and not absolute user figures. The dependency graph method presented later allows one to obtain absolute usage number, so the differential form is not further used.

The interpretation of the variables and parameters and their dimensions is presented in table 2.1. When applying the formula one should note that it is non-linear, so ordinary linear regression can not be used. Instead one can use non-linear least squares regression, but note the correct number of the degrees of freedom. This can be done using existing mathematical/statistical packages. For this thesis the `NonlinearModelFit` procedure in Mathematica was used.

Chapter 3

Dependency graph analysis

Figure 3.1: Example of a dependency graph
Source: Own work, created using Graphviz



No software project stands entirely on its own. Software is usually developed by taking one or more existing libraries of components and combining those components in ways to create new products. Take for example a simple chat application. The chat application uses a library for user interface development that provides components such as a window a text entry field and a button (that is labelled "send message" by the chat application). This user interface library in its turn uses a graphics library to draw the lines, rectangles and text necessary for the fields and buttons. The graphics library uses a library to read font files and use the fonts to turn text into pictures that can be displayed on the screen. The graphics library then sends the contents of the window to the window manager, which in turns uses graphics card driver to instruct the hardware. The chat application uses a networking library to provide it with the basic components for internet communication and uses a file library to store the users settings. The same file library is also used by the font library to read font files. The dependency graph so described is drawn in figure 3. Compared to a real chat application the graph is hugely simplified, tracing a real chat application back to all the components involved will likely result in hundreds of libraries used.

In the remainder of this thesis the terms 'project', 'package' and 'library' will be used as synonyms for a node in the dependency graph.

3.1 Dependees are adopters

In the example from figure 3 the font library and the chat application use the same library to access files, but this need not be the case. There can be several competing libraries implementing similar functionality. It could even be the case that the font library and the chat application use a different library, effectively meaning that *both* libraries are required to use the chat application. This might seem wasteful, and in a certain sense it is, but it is common practise and there is a good reason to it.

A project developing a file reading and writing library has as its target audience all project that require such functionality. This target audience has a free choice in whether they use the projects implementation or a competing implementation. Given that in the open source community there are no license fees, the selection happens on, for example, technological merits and social factors. The selection also depends on the problem at hand, the font library may have a very good reason to choose a specific file library, while the chat application may have equally good reasons to choose a different one. The two libraries will happily co-exist, each having their own niche.

The relation between software projects and dependency relations can be considered as one of technologies and adopters. Each software project has a defined problem it provides a solution for. The solution it provides, and hence the project as a whole, could be considered a technology. When a project uses another project to solve a sub-problem, they are effectively adopting its solution. The dependency graph can therefore be directly reinterpreted as a graph of technologies and adopters.

Since the number of dependees of a package is interpreted as the number of adopters one would expect from theory that it follows a Bass model growth. This hypothesis will be tested in the empirical part of this thesis.

3.2 Henderson-Clark patterns

The Henderson-Clark patterns from section 2.1.2 depend on the context in which they are applied. In the dependency graph they are interpreted in the context of an individual package which uses other packages as its components. An alternative perspective would be to consider the way in which the technology is implemented within a package, the Henderson-Clark patterns would then get an interpretation that differs from the one presented here.

Component change With component a small part of the packages changes significantly, at least internally. For example a package moving from using one library to implement a technology to using another library to implement the same technology would be a component change. In terms of the dependency graph the component change would be noticable as a change in the dependencies of a package. This could be swapping one library for another, or adding a library or removing one.

Architectural change With architectural change, the outward appearance of the package changes radically. That is, the way the package interacts with other packages changes in a way that renders previous interfaces between them

incompatible. The dependees of the package that changed the architecture need to be adapted before they can interface with the package again. In terms of software engineering this is an application interface change, more specifically an application programming interface (API) change if the code of the dependees needs to be changed or an application binary interface (ABI) change if merely recompiling the dependee suffices to resolve the interface conflict.

In terms of dependency graphs, an architectural change can be interpreted as an change in the package that requires change in its dependers. This can be detected by the following effect: The package updates and if the package uses a major-minor versioning scheme the major part is incremented. The dependers need to implement the change and until they do so they are incompatible with the new version. Such an incompatibility manifests itself in an dependency which requires a specific version of a package, for example package `kdelibs` version 3.5.9 requires `qt` with version less than 4. Once the depender has updated its code the version specification usually flips, for example, the updated `kdelibs` version 4.0.0 requires `qt` version *at least* 4.

Dependencies with explicit *less than* version specifications can only appear if an architectural change has occurred in the dependee. If the change where not architectural, the outward interface of the package would remain compatible with the previous version and the version's specification would be unnecessary. Dependencies with a *greater than* version specification are not necessarily indicators of architectural change in the dependee. The depender might for example require a version where a certain prohibitive bug is fixed or where a required feature is added.

Incremental innovation In incremental innovation the packages under consideration undergoes a minor improvement which does not alter the architecture or composition of the package. Trivial software examples of incremental innovation are higher performance or reduced size in some respect. But one can also argue that bug fixes or even minor feature additions are incremental innovations since they do not change the composition or architecture of the package. Therefor the concept of incremental innovation can be roughly translated to the concept of a minor release in software engineering.

In terms of dependency graphs an incremental innovation of a package results in a new version of the package, but no component change or architectural change. The absence of component change has the consequence that the dependencies of the package do not change with the new version. The absence of architectural change has the effect that there will not come any less-than version dependencies with the old version of the package as dependee. In short, an incremental innovation can be recognised as a change in version number, without any changes in the dependency graph.

Architectural innovation With architectural innovation the set of components used to create the product is changed very little, but the manner in which they are composed is changed such that the outward appearance of the product changes. Examples in software engineering are refactoring the public interface, incompatible changes in communication protocols and storage formats shared with users and .

Modular innovation Modular change is when a project changes its own dependencies without consequences for the projects depending on it. This is the case when a project decides to adopt a new technology, for example a new multimedia format, or when the project moves from one implementation of a technology to another.

A modular innovation can be recognised in the dependency graph by a change in a package's dependencies without a change in the package's dependees.

Radical innovation Radically changing the architecture of a package is quite uncommon, once a project has settled on a certain overall structure this structure changes only incrementally. Radical architectural innovations are not favoured by the dependers, since it entails a radical change in the way they interface with the package and such changes are often very laborious. In fact, there are examples of packages that went to a moderate architectural change and where then forked by their dependers; the dependers would rather choose to maintain the old version themselves than adapt to the new architecture! Radically changing the architecture of a package is almost certain to eschew all the dependers.

Radical architectural innovation is therefore usually accomplished by starting a new project with the new architecture in mind. It is not uncommon for some developers of an old project, fed up with the old architecture, to start a new project, based on the lessons they learnt while developing the old project. The new project can then be developed in peace, other developers and dependers may decide to switch from the old project to the new project and given enough time, everyone will have switch to the new architecture. Or if the new architecture proves unsuccessful, they keep continue with the old architecture. This is where the evolutionary nature of open source development comes in.

Chapter 4

Gathering real-world data

In this chapter the methods developed so far, analysing dependency graphs, will be applied to real-world data. First a list of available datasets is compiled, then a particular set is selected, collected and processed into a dependency graph changing over time. This provides the input for the methods described in the chapter 3. In chapter 5 some of these methods will be applied.

4.1 Qualities of a dataset

To apply the techniques of the previous chapter one would need accurate project dependency information over time. From a scientific point of view there are three qualities a dataset should have to produce good and relevant innovation information that can arguably be generalised to the open source community as a whole. And then there is the practical aspect as well.

1. The list of projects. The open source community contains a huge amount of projects, this will be quantified in the next section. These projects differ greatly in their size and their relevance to the open source community as a whole. Ideally one would like to include every single project in the dataset, but this is impossible in practise since smaller and less relevant projects are likely less known. Projects that cater only to a niche subject (for example tools for specific exotic hardware) may not even be known outside a small circle of users. It can therefore be concluded that the complete list of open source software projects will never be known. The unavailability of this list prevents a pure random selection from being made. A good dataset should contain as many projects as it can (completeness) with as much heterogeneity as it can. Or, put differently, with as little bias as it can (neutrality).

Huge list of projects can be obtained by searching the internet for the words "open source" or other tell-tale terms and collecting all the pages that appear to be open source projects. This would however create false positives, through several mechanisms: a) the method can incorrectly identifying a page as being a projects, where in reality it is for example a news article about a project. b) projects can have mirrors, where people create an identical copy of the project to increase the availability. c) the projects may be "development trees" where a developer or group of developers make a copy of project to try out new ideas before integrating them in the main project. The Linux kernel tree is a known

to use this model, where individual developers make their contributions to their own copies of the entire projects, which are then collected and ultimately integrated into the main project.

The matter of the false positives is further complicated by the existence of 'forks' as described in 1.3. Here a project is intentionally copied to add new ideas, much like the "development trees", but this time not with the intention of contributing back to the original project, but rather to start a new project on its own. The distinction is between a development tree and a fork is mainly a matter of intent and is therefore subjective. When a fork is the consequence of a developer split or disagreement its existence is not only subjective, but also controversial and debated.

In conclusion: A complete list of projects is impossible to determine and creating an approximate list is likely to be subjective and even controversial at times. If the list contains a bias towards some area of development than the conclusions drawn from it will also be biased towards that area. It is possible that the list contains some noise where matters become more subjective, where to draw the line between the same and similar but different projects and when to include or ignore a small or unused project. It is assumed that these errors are small enough and uncorrelated enough to not make a difference in the large scale structures and statistical averages that are under investigation in this thesis.

2. The dependencies. Once the list of projects is clear the dependency relations between the projects can be mapped. Again, one can question the false-positives and the false-negatives. In section 4.2.3 a list of 'distribution package databases' will be introduced. These databases are used to install and operate the software. For example, if software package A requires package B to run and the user requests A to be installed, the system will use the database and install both package A and B. If the database contains a false-negative than it would not know that A requires B, it would neglect to install package B and consequently package A would not operate properly. The system would effectively fail. It is therefore necessary for the correct operation of the system to have no false-negatives. A false-positive would have the consequence of installing unnecessary packages (also known as 'garbage' among the users of such systems). This is wasteful on resources and can lead to problems when the installed unnecessary package is so faulty that it harms the system as a whole. Both these consequences lower the user experience of the database and there is therefore a competitive pressure to keep the number of false-positives to a minimum. The observation is thus that in decent distribution package databases the false-negatives are practically absent and the false-positives are kept to a minimum. There are therefore a number of very high quality databases of package dependency information available. This observation was one of the primary motivations behind exploring the method of dependency graph analysis.

3. Chronology of the data. The dataset should not only contain a list of projects and dependencies, but also track how this changes over time. To analyse how a project gets adopted by other projects the growth of new dependency relations needs to be quantified. Scientifically, the interesting qualities are resolution and timeliness. Resolution refers to the minimal discernable unit of time in the dataset. If the dataset is produced by taking monthly snapshots of the projects and their relations, then the resolution would be one month. Timeliness refers to the responsiveness of the data gathering method to changes. When a project or dependency gets added or removed, will this be apparent in the next

snapshot? Or will it take a while for the information to trickle through. As will be shown, not all distributions package databases are set on including the very latest packages, they rather wait until they are tested and possible bugs are fixed.

4. Practicality From a practical point of view it is advantageous if the dataset is easily available and codified. Easily available means that the dataset can be obtain by reasonable means, for example by downloading from one location. Codified means that the data is written in a manner that allows automated processing. For example, taking all the projects homepages as the data source would not satisfy both. Even though this might be the most accurate method, it is infeasible to download all the homepages and very difficult to extract the dependency information.

4.2 Sources of data

Given these qualities an investigation can be made of the data sources available. There are three large bodies of data sources. First, one has the project hosts. These are organisations that provide facilities to open source projects such as a website, a mailing list a source code repository, etcetera. As an extra service and promotional tool they provide lists of projects hosted on them, often including much meta information on the project, such as its rate of development, the number of developers, activity by month. This information gives so many insights in open source development that it warrants a separate research project. In fact, this has been done in the work by Adams et al. (2008) amongst others. Second, one has the project directories. Much like a telephone directory these are manual or automated efforts of indexing open source projects. Their purpose ranges from helping users find a specific project to collecting extensive amounts of statistics and metadata. The third category of data sources are the distribution package databases. These are used by the open source distributions to track what other packages need to be installed in order to use a certain package.

4.2.1 Project hosts

Table 4.1: List of FOSS hosts and their reported sizes.

	Developers	projects
GitHub	345,000	1,061,000[a]
Assembla	180,000[b]	
SourceForge	¿2,000,000	246,790
Bitbucket	58,100	31,356
Launchpad	1,169,086[c]	18,901
CodePlex	151,782[d]	16,917
Project Kenai	53,192	12,132
Gitorious	?	9,532
Google Code	?	5,675
BerliOS	47,778	5,467
GNU Savannah	49,155	3,242
TuxFamily	?	2,279
BountySource	?	1,337
Alioth	10,187	896
Tigris.org	?	686
Eclipse Labs	?	623
KnowledgeForge	1,039	266

[a] GitHub encourages forking as a form of branching, a search query for projects that are forks showed that at least 31% of the projects are forks.
[b] Figure according to the homepage, the user list only has 3,086 public users.
[c] This is the primary host for Ubuntu development. It is likely that Ubuntu users are encouraged to register as a Launchpad developer, thus inflating the number.
[d] Data from Wikipedia, could not be verified.
Source: own creation, list from Wikipedia, data from respective websites in 2010.

A survey of general purpose open source project hosts has resulted in table 4.1. The list of project hosts was first retrieved from Wikipedia. For each host in the list a search was done to find the number of projects hosted and the total number of developers registered at their platform. None of those figures have double-counting when developers work on several projects, but developers may have registered several accounts.

A particularly large number of projects is hosted on GitHub, but projects on this host tend to have a high number of very similar clones. GitHub encourages a development method where developers make a personal copy of the entire project and apply their ideas there first, before integrating them in the main project. Integrating back into the original project is not a necessary part, developers can maintain their version as a fork. All these copies inflate the number presented by GitHub. A query for the tag 'fork' which is added to such projects reveals that at least 31% of the projects are copies of other projects.

Large numbers of developers can be found on SourceForge and Launchpad. The former is one of the first and most well-known hosts, hence has gathered many projects and users over the years. The later is developed Canonical, the

company behind the Ubuntu Linux distribution. Users of Ubuntu are encouraged to register and contribute.

A rough idea of the size of the open source community can be obtained from these data. Developers can register at several project hosts and will have to when they want to work on projects from different hosts. It should therefore be assumed that there is a lot of overlap in the user base. Assuming maximal overlap, a lower bound of the total number of developers is two million, the highest number of developers from a single host. Compare this with 913,100 software developers from the U.S. Bureau of Labor Statistics.[1] This would lead to the result that the open source software development workforce is at least twice as large as the entire US software development workforce. But that would be a rather quick conclusion, there could be a lot of duplicate or inactive accounts on SourceForge and even the active users are unlikely to spend the equivalent of a full-time job on an unpaid project. So The actual number of working hours spent will be less impressive. On the other hand, there are many companies investing paid hours on open source projects. From an innovation perspective the total amount of available specialist knowledge among the developers might be as interesting as the number of working hours. It can be argued that the former scales more with the number of 'heads' than the number of hours.

Reflecting on the qualities a dataset the list of projects is very extensive compared to the other sources, a bit too extensive. According to a 2007 study by the FLOSSMetrics project only 8% to 10% of these projects are active, the rest has not shown activity recently.[2] It is unlikely that the majority of these abandoned projects where once major open source projects, most of them are rather insignificant and could be considered noise on the list.

On most hosts anyone can register for a free account and start a new project, therefore no selection bias is expected from these hosts. In practise specific groups of open source developers can cluster around a specific hosts. Unsurprisingly the open source projects initiated by Google tend to be on Google Code and Ubuntu related project tend to be on Launchpad.

The presented list does not include hosts that cater to a specific project or development environment. As an example freedesktop.org caters to projects related to graphical desktop components and Java.net, LuaForge and RubyForge cater specifically to projects developed in the Jave, Lua and Ruby languages. The existence of these dedicated hosts implies that they will be under represented in the general purpose hosts. Whether that is a significant bias is difficult to determine.

To get a complete and unbiased picture one would need to combine all the general purpose and dedicated hosts. As mentioned the both kinds suffer from biases in their list of projects. Furthermore, large, significant projects, which should be included in the list to get a complete picture, can be in all the project hosts.

Many of the larger projects run their own hosting facilities and are therefore missing from the list of projects. Examples of such projects are the Apache webserver, OpenOffice, GNOME and KDE. By their nature these projects are large, famous and much used, not having them in the list of projects is a detrimental bias.

[1] `http://www.bls.gov/ooh/Computer-and-Information-Technology/Software-developers.htm`

[2] `http://robertogaloppini.net/2007/08/23/estimating-the-number-of-active-and-stable-floss-projects/`

The second quality, dependencies, is even more difficult. The project hosts do not require their projects to list which other projects they depend on because there is no reason to. The hosts also do not have a clear benefit from having such a list. Gathering the dependency information would therefore mean looking at the individual projects and finding out its dependencies, which is not codified in a standard way. Doing this for hundreds of thousands of projects is near impossible.

The idea of gathering data from project hosts is quickly abandoned at this stage.

4.2.2 Data from project directories

Table 4.2: Overview of FOSS directories.

	Projects indexed	provided meta data
Ohloh	437,710	popularity, versions, commits, code measures, developers, geographic location, etc...
Freshmeat	42,445[a]	popularity, versions
IceWalkers	3,107	latest version
OSSdirectory	855	none
FSdirectory	6,650	versions

[a] Also includes some commercial packages
Source: own creation, list from Wikipedia, data from respective websites.

The survey of project hosts also resulted in what could be called 'project directories'. These interesting data sources try to create a list of projects and do some measurements on them, much like what this thesis aims to do. Particularly interesting is Ohloh, which has indexed many of projects and measures a lot of information that is valuable for innovation research. Ohloh tracks the individual changes made to a project and for each change it knows who the author is, what programming languages are used and how large the change is, in terms of lines of code. This is then used to create popularity measures, activity measures and size measures, for both the projects and their developers.

The directories provide a good list of projects with interesting and relevant information, but no dependency information. The directories only contain meta information on the projects and point to the project's website for more information. In particular the dependencies are not mapped in any way. So these data sources are also unusable for this thesis.

4.2.3 Data from distribution package databases

Table 4.3: Major FOSS distributions and their package databases.

	Initial release	package manager	source/binary	packages	release cycle
Ubuntu	20-10-2004	dpkg	binary	34,580	$\frac{1}{2}$ year
Fedora/Red Hat	13-04-1995	rpm	binary	7,334	$\frac{1}{2}$ year
(open)SUSE	01-03-1994	rpm	binary	6,011	1 year
Debian	17-06-1996	dpkg	binary	40,163	2 year
Mandriva	23-06-1998	rpm	binary	5,779	$\frac{1}{2}$ year
Mint	27-08-2006	dpkg	binary	30,000	$\frac{1}{2}$ year
PCLinuxOS	01-10-2003	rpm	binary	?	$\frac{1}{2}$ year
Slackware	16-07-1993	pkgtools	binary	10,590	1 year
Gentoo	31-03-2002	portage	source[a]	14,018	rolling
CentOS	14-05-2004	rpm	binary	2,599	$\frac{1}{2}$ year
FreeBSD	01-11-1993	ports	source	22,020	$\frac{1}{2}$ year

[a] About 100 packages are also provided in binary form.
Source: own creation, list from Distrowatch, data from Wikipedia and respective distribution websites.

There are numerous distributions of open source software. So many that it provoked Ladislav Bodnar to start a website maintaining a comprehensive list of all distributions available. At the moment of writing the database contains 665 distributions, 320 are labelled as active, 50 as dormant and 295 as discontinued. The majority of these distributions are very small projects or subtle variations of other projects. His website Distrowatch also maintains a list ranked by popularity. This list is presented in table 4.3 and augmented with distribution statistics collected from Wikipedia and the distribution's websites.

For each distribution a number of details is given in the columns. The date of its first published version is given. The 'package manager' revers to the system used to install and maintain software in a running system. This is relevant for the thesis since the package manager must know what packages depend on what other packages so that it installs all the prerequisites. The 'source/binary' column specifies whether the packages are installed by compiling the source code or if the distribution provides pre-compiled binaries. The later case is more popular since compiling software is very slow and error prone compared to simply copying pre-compiled files. The last columns list the number of packages in the package database at the time of writing and the average duration between new versions of the package database. Gentoo is unique in that it does not periodically releases a new version of the package database, but updates its database as new versions of packages become available.

It should be noted that some distributions are variations of other distributions, even within this list. Mint is based on Ubuntu which in turn is based on Debian and PCLinuxOS is derived from Mandriva. See also figure 1.2. Fedora is a commercial Linux Distribution developed by Red Hat, unlike the other distributions it is not free to download. The open source license requires them to publish some of their source code, which anyone can compile to build their own free version. CentOS is a distribution that does exactly this and is therefore based on Fedora. Interestingly, the for-money Fedora is more popular in this list than its free-of-charge clone. Explanations can be the support Red Hat provides, it's proprietary extensions and branding.

Reflecting on the qualities of a dataset the numbers of packages do not seem too impressive at first sight. But unlike the previous datasets there are few insignificant packages, no distribution would like to spend time supporting a package that is unfinished or abandoned by its original developers. On the other hand, their user base requires them to include all the popular projects. Given that the most relevant packages are included, the fact that the list does not include 'noise' in the form of insignificant projects could be considered a bonus.

Still, there can be a bias in the selection of packages. Ubuntu, for example, has a specific target audience of novice Linux users and provides them with a Gnome based desktop environment. This means there is less pressure incentive for Ubuntu to include other desktop environments and advanced server software in their package database. Despite this, Ubuntu's database still includes a few other desktop environments and various advanced packages. Broad support appears more important than the burden of supporting many components. In general, the distributions have a user base that is very broad, some distributions are used from large server farms to mobile device and provides software packages for users that range from children to professionals in various fields.

A distinctive advantage of distribution package databases is that they contain very good dependency information. As mentioned earlier the package database needs to contain correct dependency for correct operation of the system and since the package manager operates automatically this information needs to be fully codified. With relative ease one can extract the entire dependency graph from the package database. This greatly facilitates gathering a dataset.

Source distributions will necessarily contain more a more complete dependency graph than binary distributions. Both kinds of package databases need to list all the dependencies required to *use* a particular package, but source base distribution need to include all de packages required to compile and use the project.

4.2.4 Conclusion

The Gentoo portage database and the FreeBSD ports databases are ideal candidates. Both have all the good qualities of a source based package database. FreeBSD's has the advantage that it has a long history and more packages, Gentoo's has the advantage that it has better time resolution. Since author is very familiar with Gentoo's database, this one is chosen.

It would be interesting to develop a hybrid approach by combining information from several sources. For example one could combine package databases with information from services such as Ohloh. Or one could develop methods to (heuristically) parse the source code of the projects and extract the dependencies from there. This method is difficult to implement, but once implemented it can collect other statistics from the source code, much like Ohloh does.

4.3 Processing the Gentoo Portage dataset

Now that a data source is selected, it is time to extract the required information and process the data into a form that allows easy calculations. The major steps are collecting the raw data, parsing this into a simpler format and producting the final dependency graph from this simpler form. Some postprocessing can then be done on the dependency graph. In the process the dataset will shrink from thirteen gigabytes taking more than a week to collect to thirty megabytes that can be processed in four seconds.

4.3.1 Collecting the raw ebuilds

The Gentoo portage database consists of a number of files, at least one for every version of every package, contained in a large directory structure. This entire structure is kept in a CVS revision control system that has tracked all changes to the database since the start of the project arround 2000.

Using the `cvs` command one can download the entire database as it was at a certain point in history. For example the following command would download the database as it was on 1 December 2003:

```
cvs -d :pserver:anonymous@anoncvs.gentoo.org/var/cvsroot co \
  -D 12/01/2003 gentoo-x86
```

Using a small utility written for the task, this command was repeatedly invoked to download all the databases from 1 January 200 untill today, with increments of one month. Of all these downloads, only the `.ebuild` and `.eclass` files are stored, the other files are ignored to save space since they contain no relevant information. This whole downloading processes took about a week and a half and the resulting database consists of three million files occupying thirteen gigabytes of space.

Later it was found that there are also `.eblit` files which are relevant, but these are just recently introduced and only used by a handful of packages. The missing `.eblit` files could therefore be introduced by hand and the week long process did not have to be redone. There are also files specifying packages renames, but since these only get appended to and never deleted they where taken from the latest tree.

4.3.2 Parsing the ebuilds

The files are written in a text based computer language called ‘ebuild’ which is based on the Bash shell script language. Being a scripting language, the files can refer to other files and include complicated code to calculate dependencies on demand. This eases the task of the database developer, since he can automate many processes, but it complicates the task of extracting data. Several approaches where tried to extract the data in the industrial quantities the analysis requires.

The first approach was to use Paludis and its C++ bindings to load a repository and extract metadata. Paludis is a package designed to process ebuild files, when then query its database interface to gather all the dependency information from all the packages. This approach takes a lat of time, it requires around a

half an hour per database, but it fails on some of the older databases because the format of the database changes over time.

The second attempt was to use a custom build metadata extraction program that also supports older version of the database. This parser looks for text patterns resembling dependency specifications and implements only a minimal amount of the ebuild file format (basically only the `inherit` statement). This technique is very fast, processing the entire set in 70 minutes, but fails on the newer databases that use complex techniques such as macro's in the dependency specifications.

The final method is a hybrid of the first two, using the Paludis' `instruo` command on the later trees to create simplified versions that do not use the advanced macro's but contain the same information. The `instruo` command was run on every copy of the database downloaded. Where the command failed on a package (mainly the older format ones) the original was kept. The total running time of this operation was around four days.

4.3.3 Producing the dependency graph

Now the data is in 154 snapshots of the package database in a simplified text based format. This is several gigabytes and several millions of files large and needs to be processed into dependency graphs. Obviously this it is inhumane to do this by hand, therefore more specialised tools were developed.

By design the database can work with complicated dependency relations, such as "package `amarok` requires package `phonon-kde`, minimum version `4.3`, but only when feature `player` is required". This is would be coded as `player? ( >=kde-base/phonon-kde-4.3 )`. A complete example of the run time dependencies for the Amarok music player is given in figure 4.3.3. The figure includes more complex rules such as "either package X or package Y is required", " package X is required to contain feature Y", etcetera.

These conditional rules are relevant when compiling and running software, but the conditions are not necessary when analysing component use from an innovation perspective. If Amarok has an optional dependency on a package, the developers of Amarok are actively using the innovation provided by that package even though it may not be enabled by the end user in the final product. For this reason, and of simplicity, all the conditionals are ignored when parsing the ebuilds.

When a database snapshot contains several versions of the same package, only the latest is used. For some of the analysis techniques presented in section 3.2 the version information is required. But at this point it was decided not to include different versions to simplify processing analysis. When several versions where available in the database at a certain point in time, only the highest version was included. This ensures that each snapshot represents the state of art at that time.

In the Gentoo Portage database the dependencies are sorted in two kinds, compile time dependencies and runtime dependencies. The first kind are required to build and install the package. The second kind is only required when actually using the package. The reason for this distinction is a technical: it solves installation issues with cyclical dependencies. For the current purposes both kinds of dependencies are considered equal.

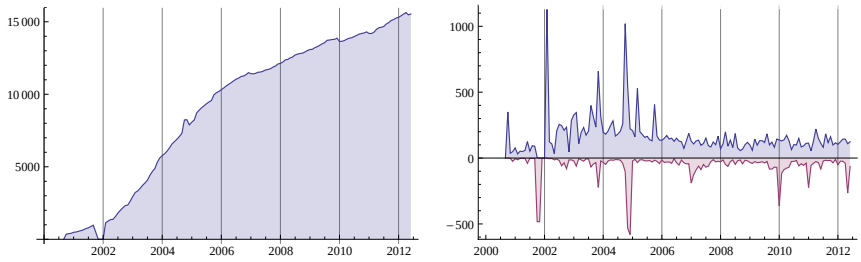
Figure 4.1: Runtime dependencies of the Amarok music player.

```
>=media-libs/taglib-1.6.1[asf,mp4]
>=media-libs/taglib-extras-1.0.1
player? ( app-crypt/qca:2
          >=app-misc/strigi-0.5.7[dbus,qt4]
          || ( >=dev-db/mysql-5.0.76
               =virtual/mysql-5.1 )
          >=kde-base/kdelibs-4.3[opengl?,semantic-desktop?]
          sys-libs/zlib
          x11-libs/qt-script
          >=x11-libs/qtscriptgenerator-0.1.0
          cdda? ( >=kde-base/libkcddb-4.3
                  >=kde-base/libkcompactdisc-4.3
                  >=kde-base/kdemultimedia-kioslaves-4.3 )
          embedded? ( <dev-db/mysql-5.1[embedded,-minimal] )
          ipod? ( >=media-libs/libgpod-0.7.0[gtk] )
          lastfm? ( >=media-libs/liblastfm-0.3.0 )
          mp3tunes? ( dev-libs/glib:2
                      dev-libs/libxml2
                      dev-libs/openssl
                      net-libs/loudmouth
                      net-misc/curl
                      x11-libs/qt-core[glib] )
          mtp? ( >=media-libs/libmtp-0.3.0 )
          opengl? ( virtual/opengl ) )
utils? ( x11-libs/qt-core x11-libs/qt-dbus )
!player? ( !utils? ( media-sound/amarok[player] ) )
!media-sound/amarok-utils
player? ( >=kde-base/phonon-kde-4.3 )
```

Sometimes, package get renamed or moved around in the database, this needs to be accounted for. Luckily, to allow users to upgrade from an older version to a newer version the processing of moves and renames has been automated in the Gentoo Portage database. The developers maintain a list of all the moves and renames that have happened in the database in a structured format. The latest version of this list is used to retroactively change all the package names in the historical snapshots to their modern name. This ensures that moves and renames do not harm historical continuity in the dataset.

Using all the observations and choices made above, a tool was developed to extract dependency graphs from all the simplified database snapshots. The shear scale of the resulting dataset, 1.3 million packages and 6.9 million dependency relations, required a solution to store efficiently. Therefore, a compressed format was developed that only stores the changes between the historical snapshots instead of storing whole snapshots. The extraction process took three hours and resulted in a 29 megabyte file. Reading out this file and performing calculations of the kind that will be needed later takes about 4 seconds. The dataset is now in a usable form.

Figure 4.2: Growth of the number of packages in the Gentoo package database.
Source: own creation using Mathematica



(a) The total number of packages over time

(b) The number of package additions and removals over time. The blue curve represents the monthly total number of additions, the red line the monthly total of removals.

4.3.4 Processing the graph

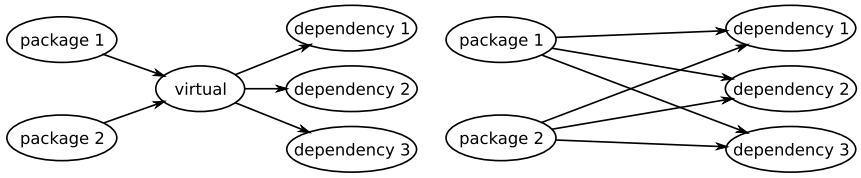
Growth of the number of packages. In figure 4.3(a) the number of packages in the database is plotted over time. One can see how the database started in 2001, underwent a period of rapid growth between 2002—2006 and settled into calm a linear growth from 2006 onwards. In figure 4.3(b) the rate of introducing and removing packages is plotted over time. This show two spikes, one at the start of 2002 and one in 2005. The cause of these spikes was not thoroughly investigated, but a likely cause is a massive cleanup and refactoring of the database. This is a warning sign that the data exactly at these points might contain a lot of noise. In general, the data before 2006 should be considered with more caution than the data afterwards.

The effect of the growth of the entire dataset on the number of dependencies for individual packages was investigated, but no influence was found. Since the total number of packages grows one would expect the 'market' for a certain package to grow and thus the number of dependency relations to that package to grow. To compensate for this one could divide the number of dependers by the total number of packages, compare this with using a market-share instead of an absolute number of users. In practise, this only made any significant difference for early data, but that was determined to be unreliable anyway. In the interest of keeping the analysis simple no compensation was made for the growth of the number of packages.

Virtual and meta packages. It sometimes happens that several projects implement the same functionality and are entirely compatible. This can be caused by forking, where the two copies maintain compatibility with each other. In other cases, such as with OpenGL or Java, entirely separate projects are started that implement the same standard. The end result is that several packages exist that provide a certain technology and that a package depending on this technology is indifferent to which package provides it.

In the Gentoo Portage database this is accounted for by introducing 'virtual' packages representing abstract technologies rather than concrete implementations. In figure 4.4(a) one can see how two packages require a technology

Figure 4.3: The elimination process of virtual and meta packages; dependency relations are connected through and the virtual package is removed.
Source: own creation using Inkscape



(a) Before elimination a virtual package represents a common group of dependencies.

(b) After the elimination all the dependencies of the virtual packages are added to the dependers of the virtual package.

provided by `virtual`. The package `virtual` does not have content of its own; upon installation it will install nothing. However, the package depends on several 'real' packages, which will be installed by the package manager. Since only one implementation of a specific technology is required the dependencies are of the "either X or Y or Z" variety.

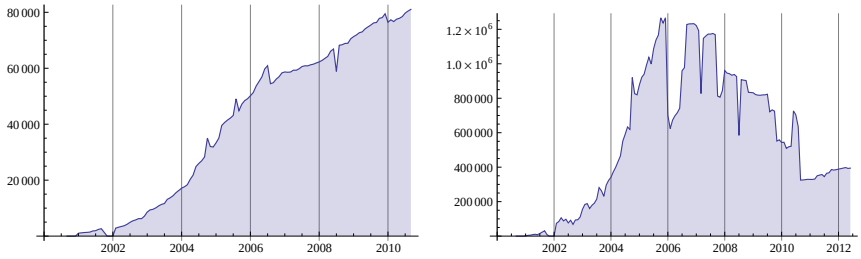
Related to this are 'meta' packages, which group a common set of dependencies. Sometimes it is useful to consider a group of packages as a single package. For example a word processor, spreadsheet and presentation package could collectively be considered an office package. These groups are mostly used as a convenience for users, it allows them to install a bunch of packages in one go. It is implemented similar to the virtual packages, but instead of an "either X or Y" condition, the dependencies are of the form "X and Y and Z" form. These meta packages are only used to convenience users and are not depended upon by other packages. They can therefore be safely discarded.

Since these virtual and meta packages are not true packages and do not represent actual open source projects a method was devised to eliminate them. As mentioned the meta packages can simply be discarded, but the virtual packages contain valuable dependency information. The packages that depend on a virtual package do so because they use the technology it provides. If the virtual is simply removed this technology is missing from the dependencies of the depender. To compensate all the dependencies of the virtual are added to all the packages that depend on the virtual. The process is explained in figure 4.3.4.

Applying the virtual elimination technique increases the number of dependencies. Suppose the original virtual has i packages depending on it and depends on o packages. The number of dependency relations before elimination is the sum $i + o$. After elimination this number is the product $i \cdot o$. If both i and o are large than the product is huge compared to the sum. The number of dependency relations over time is plotted in figure 4.5(b). After the virtual packages are eliminated using the method described the number of dependency relations increases to the amount plotted in figure 4.5(b). The increase is tenfold! After processing the majority of the dependency relations are caused by the elimination procedure, completely swamping the original data. Even worse, the number of dependency relations over time is now highly erratic, containing huge rises and falls.

In the interest of keeping it simple it was decided not eliminate the virtual

Figure 4.4: Growth of the number of dependency relations in the Gentoo package database and the effect of eliminating the virtual and meta packages.
Source: own creation using Mathematica



(a) The number of dependency relations in the unprocessed dataset over time. Note the strong relation with the number of packages, their fraction converges to around 4 dependencies per package.

(b) The number of dependency relations in the virtual eliminated dataset over time. The number of dependencies increases tenfold and shows highly erratic behaviour.

packages. This does not affect the outcomes significantly for a two of reasons. First, the majority of the dependencies created by eliminating virtual packages happen in a few specific areas. By not considering packages in these areas one avoids the influence of virtual packages, at the cost of not being able generalise conclusions over those areas. Second, the virtual dependencies are of the "either X or Y" kind, so in practise only one of the packages is installed. Thus the number of dependency relations in practise is only $i \cdot 1$ instead of $i \cdot o$. Thinking about this, the virtual packages represents an abstract technology that is depended on. The packages themselves are ignorant about the specific implementation used, they operate in the abstract as well. Keeping the virtual packages as they are more closely resembles reality than the elimination procedure presented above.

Chapter 5

Analysing the real-world data

In this chapter the real-world data gathered in the previous chapter will be analysed using the method developed in chapter 3. In particular the Bass diffusion model from section 2.1.3 will be used to model the adoption of technology by projects as it is represented by packages gaining dependencies on other packages. But first, graph clustering analysis is applied on a small but interesting subset of the entire graph.

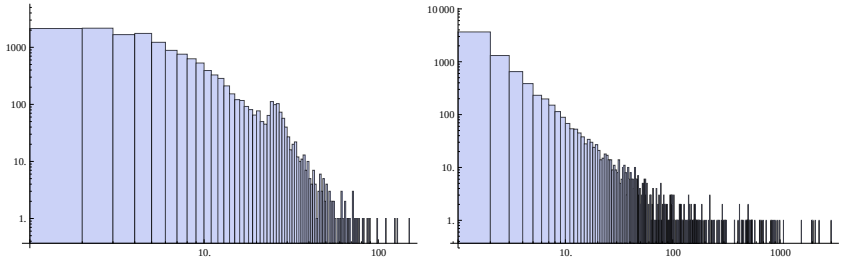
5.1 Exploring the last snapshot

One of the first thing attempted after generating the dataset was to visualise the entire graph of the latest snapshot. The problem is, to make any sense of a graph it has to be laid out, nodes that are connected should be placed close to each other so that connecting lines are short and have little overlap. Software packages such as Graphviz, Tulip, Gephi, Jetty and Cytoscope have been tried, but after days of trying and many hours of calculation, none where able to produce any insightful layout for the sixteen thousand nodes and hundred thousand relations.

Since it was impossible to get a visual overview of the entire dependency graph, its structure was plotted using histograms. Figure 5.2(a) is a histogram of the number of dependencies per package. A log-log scale was required to make the plot insightful, this reflects the fact that there are many packages with only zero, one or a few dependencies and a few packages with a lot of dependencies. Statistically this means the distribution of the number of dependencies has a fat tail. Likewise, the number of dependers for each package is plotted in figure 5.2(b). This can be interpreted as the distribution of the number of adopters of a given technology. Again, there is a fat tail, even fatter than the one from the number of dependencies. The approximate linearity of the histogram suggest a power-law like distribution of the number of adopters for a given technology. According to Shalizi (2007) this is not sufficient proof and further statistical tests are required. Unfortunately, the tests mentioned in Shalizi (2007) are not easily available in Mathematica.

The entire graph might be difficult to visualise, but a small part should be

Figure 5.1: Histograms of the number of dependencies and dependers per package. Vertically the number of packages is plotted logarithmically, horizotanly the number of dependencies or depender relations is plotted for those packages, also logarithmically.
Source: own illustration, created using Mathematica.



(a) Dependencies per package (b) Dependers per package

easier. The problem with choosing a small part is that the part must have a meaningful boundary, a random selection will likely have few relations and miss some key packages. It was therefore decided to only pick the packages that belong to the KDE desktop environment. The primary reason is that the author is familiar with this set of packages and knows its structure in some detail. A secondary reason is that the packages are developed by a tightly connected community where component reuse among the projects is stimulated by creating libraries. The selection was implemented by considering only packages in the category `kde-base` from the Gentoo Portage database.

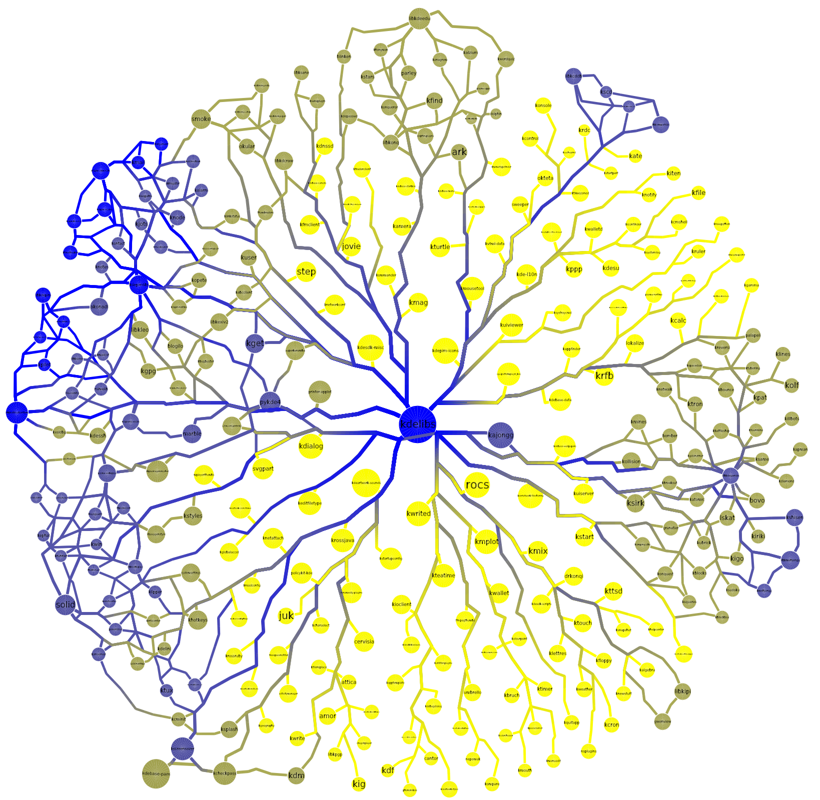
The program Tulip was used to visualise the graph, the result is in figure 5.1. First the graph was laid out using a force based method, this clusters packages that are strongly connected close to each other. Then the graphs where coloured according to their k-core measure, this is a measure for the 'connectedness' of a package. At this point there was still an unclear mess of lines between the packages, this was resolved by bundling the edges. Edge bundling merges neighbouring lines to a single thicker line, this creates a vein-like structure.

In figure 5.1 one can see that all the packages depend on `kdelibs`, the large blue dot in the middle. The `kdelibs` package provides a lot of basic functionality, such as a unified set of icons, file open/save dialogues and less visible standard components. Almost all the packages in the KDE set require one or more of these components. It should be stressed that there was no manual work involved in the layout of this graph, Tulip was able to determine using only objective, determinist mathematical methods from graph theory that `kdelibs` plays a central role in the KDE technology.

The second thing to notice are the clusters that form along the edge of the figure. All these clusters represent related areas of technology within KDE. The brownish-grey cluster immediately at the top contains mostly educational software and a few file utilities. Going clockwise, the little blue cluster next to it contains programs for compact discs. The large brownish-grey cluster on the right consists exclusively of games and supporting technologies. The complex mesh that starts around seven o'clock begins with technology used to allow users to log in. It then proceeds towards hardware related technology and desk-

Figure 5.2: Internal dependencies of modules in the KDE project. Colour represents the *k*-core measure. The graph edges have been bundled to improve readability.

Source: own illustration, created using Tulip.



top infrastructure. The big blue dot marked 'solid' at eight o'clock is KDE's hardware abstraction layer. At nine o'clock the big blue dot represents the notification library, used to notify users of hardware events ("battery low" and the likes), appointments or incoming emails. The mesh now shifts towards personal information management at ten o'clock. These contain utilities such as an email client, a note taking application, a chat client and a calendar application and related technologies. Lastly, the small brown-grey cluster at eleven o'clock contains technology to allow integration of scripting languages.

Scattered throughout the figure are yellow dots containing packages that are only connected to `kdelibs`, without any apparent pattern in their location. This was expected since the packages only depend on `kdelibs` and are not depended upon by other packages. This means there is no information that brings any insight in their nature and where to cluster them. Perhaps if dependencies from outside the KDE subset where included the packages would form more clusters.

It is remarkable how only a few dependency relations provide sufficient clues for the clustering algorithm to automatically find related areas of technology. Similar but faster clustering techniques where used on the whole snapshot with similar results. Related packages for certain programming languages (Perl, Php, Java, Python, Ruby) would cluster and packages related to either KDE or GNOME would cluster, among many more. Unfortunately the analysis software was struggling with the size of the dataset and the full set was not investigated further.

5.2 Fitting the Bass innovation diffusion model

The previous section only considered a single snapshot, but the dataset contains an entire history, so a time series based analysis can be applied. The time-series under consideration is the number of dependers (adopters) of a given packages (technology) at a given point in time. As explained in section 2.1.3 one expects this to follow a Bass diffusion curve, given by equation (2.3). This is a non-linear equation and can be fitted using non-linear least squares regression methods. The method used was the default method implemented in Mathematica's `NonlinearModelFit`. To help the method converge a manual initial guess was provided, but this process can probably be automated. To limit the search the constraints that $p \geq 0$ and $q \geq 0$ where provided. Once a least-squares fit is found its goodness-of-fit is analysed, the parameters are extracted and the result is plotted.

The goodness-of-fit analysis is done using analysis of variance, presented in an ANOVA table. The goodness-of-fit is calculated using the adjusted coefficient of determination, $\bar{R}^2$, which is the percentage of variance explained by the model corrected for the degrees of freedom of the model. This measure may not be appropriate in the context of time-series, where individual measurements are not independent, so the goodness-of-fit numbers are not to be considered fully rigorous. Different methods from Mathematica's algorithms where evaluated, such as Kolmogorov-Smirnov, but no methods where found that can be used to rigorously measure the goodness-of-fit of a non-linear model to time-series data. Escanciano (2006) proposes a new method to do such goodness-of-fit tests, mentions that bootstrapping methods are commonly used in this context and the wild bootstrapping method is the most relevant for this situation. The

paper by Escanciano (2006) and some other relevant papers where relatively recent, published in a statistics journal and highly technical, this hints that the present goodness-of-fit problem is still an active area of research. Given the exploratory nature of this thesis, absolute rigour is not crucially important at this stage and the issue was not investigated further.

The parameters are extracted from the fit and confidence intervals are calculated under normality assumptions. In a table the four parameters are presented with their mean value, their standard error and a 95% confidence interval. Since normality is assumed the confidence intervals ignore the $p \geq 0$ and $q \geq 0$ constraints.

The plot is drawn using a thick red line for the model and shades of red for the prediction bands. The thick red line is drawn according to equation (2.3) with the means values used as parameters. Then single value predictions bands are calculated for 90%, 95%, 99% and 99.9% confidence and are drawn in progressively darker shades of pink. They represent a prediction for where a single additional value would likely fall. According to the model and the uncertainty introduced by the fit, there is a 90% chance that it will fall in the innermost band, 95% chance that it false in the second band, etc. If the data fits the model properly, one would expect to see 90% of the points in the inner band. Finally, the empirical data points are plotted as black dots, connected with a thin vertical line, the residual error, to the model.

5.3 Example of imitator driver growth

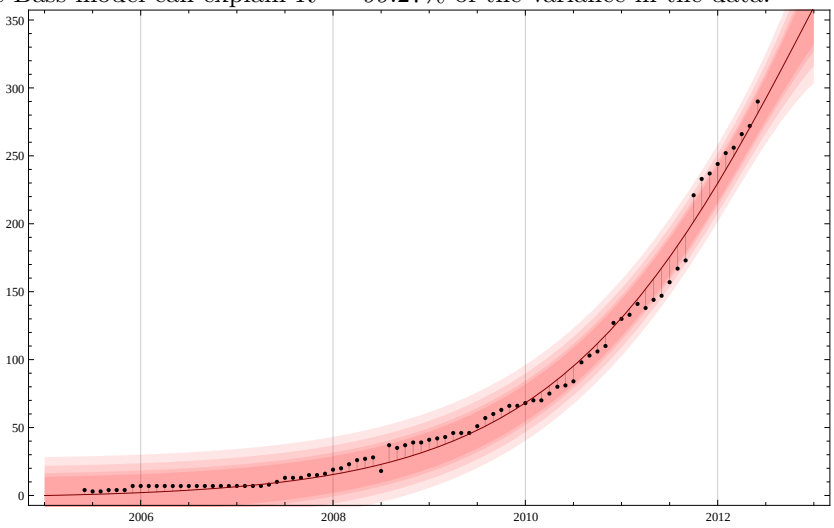
From the entire list of packages a few well-known (at least according to the author) packages where selected. The selection criteria where that the package must not have existed much before 2004, because the Gentoo Portage database was still too immature then, and the package must have gained a considerable number of dependers since its introduction. First a typical imitator driven growth and a typical innovator driven growth is presented, then eight cases are presented and finally a special case of growth and demise is presented.

The first package to `git`, a modern revision control system that shows an imitator driven adoption. It's growth can be seen in figure 5.3, the corresponding statistics are in table 5.1. The package first appeared just before 2005, it had around ten packages depending on it in 2006, twenty in 2008 and is currently used by almost three hundred packages. According to the Bass model it will continue to grow to approximately 750 users. The innovator inflow is only 0.2% of the potential market per year so one would expect $0.002 \cdot (750 - 300) = 1$ user to adopt `git` out of shear innovation. Taking the analogy with persons, if someone from the 450 current non-users where to meet a random person from the entire 750 market there is a $\frac{300}{750} = 40\%$ chance of meeting a user which can convince him to start using `git`. The chance of this happening is the imitator inflow $q = 0.73$. Therefore, the total number of users `git` can expect to gain from imitation this year is $450 \cdot 40\% \cdot 0.73 = 131$. Very much imitator driven!

The relative slowness of `git`'s growth and its dependence on imitator can be explained. First, the purpose of revision control systems needs to be explained. Open source projects, and software project in general, consist of numerous large textual files containing source code. Changes made in one place can hugely and unpredictably affect other places. To complicate matters further, usually more

than on developer works on the source code at the same time. Revision control systems track who changed what and when in the source code and even allows developers to revert changes. The revision control system in effect memorises every version of every file in the project with detailed information on who created it. This is a piece of infrastructure that is almost exclusively targeting software developers. There are competing systems such as `cvs`, `subversion`, `mercurial`, etcetera., but the basic functionality of maintaining version is provided by all of them. Thus two explanations can be derived for `git`'s growth: (1) the revision control system is not a part that affects the products delivered by the open source project and (2) there is little incentive to switch unless the new revision control system is proved to be superior.

Figure 5.3: Bass fit on the number of packages depending on the package `git`. The Bass model can explain $\bar{R}^2 = 99.27\%$ of the variance in the data.



Source: own creation using Mathematica.

Table 5.1: Results of fitting the Bass model to the total number of packages depending on `git`. Fitted using nonlinear least squares regression analysis. See figure **??** for a plot of the data and fitted model.

(a) ANOVA Table

	D.F.	sum of squares	mean square
Model	4	$9.16 \cdot 10^5$	$2.29 \cdot 10^5$
Error	81	$5.23 \cdot 10^3$	65.0
Uncorrected total	85	$9.22 \cdot 10^5$	
Corrected total	84	$5.13 \cdot 10^5$	
Adjusted coefficient of determination $\bar{R}^2 = 99.27\%$			

Source: own creation using Mathematica.

(b) Parameter estimates

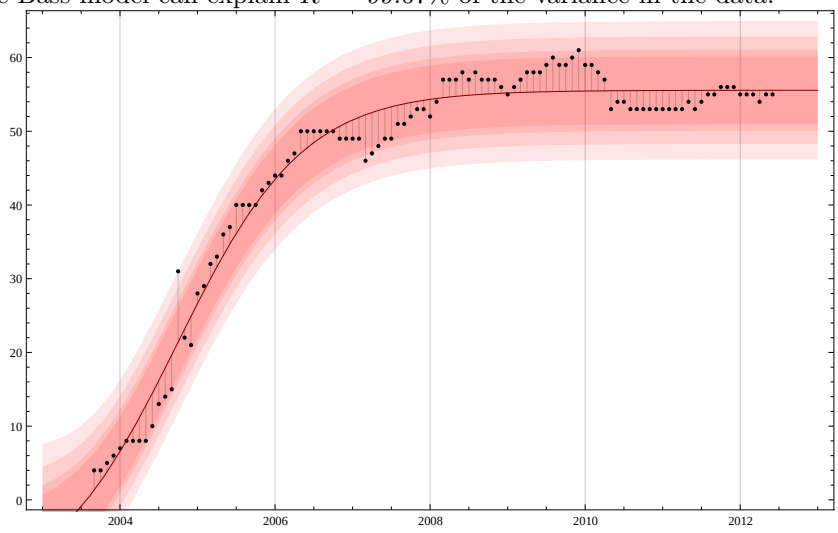
Parameter		unit	mean	std. error	95%-interval	
Start	t_0	year	2,005.03	1.36	2,002.32	2,007.73
Size	M	adopters	746	198	352	1140
Innovator inflow	p	year^{-1}	$1.97 \cdot 10^{-3}$	$2.05 \cdot 10^{-3}$	$-2.10 \cdot 10^{-3}$	$6.04 \cdot 10^{-3}$
Imitator inflow	q	year^{-1}	0.732	0.0633	0.606	0.858

Source: own creation using Mathematica.

5.4 Example of innovator driver growth

A typical example of innovator driven growth is given by `libmad`. The model is fitted resulting in figure 5.4 and table 5.4. Again, the data is neatly explained by a Bass diffusion process, in particular the rapid steep growth and the stable user base afterwards. The name is an acronym for "library for MPEG Audio Decoding" and the package provides a high quality mp3 decoder for use in multimedia applications. This might also explain the rapid growth of its adoption: multimedia applications can benefit a lot from good quality mp3 support.

Figure 5.4: Bass fit on the number of packages depending on the library `libmad`. The Bass model can explain $\bar{R}^2 = 99.67\%$ of the variance in the data.



Source: own creation using Mathematica.

Table 5.2: Results of fitting the Bass model to the total number of dependers of `libmad`. Fitted using nonlinear least squares regression analysis. See figure 5.4 for a plot of the data and fitted model.

(a) ANOVA Table

	D.F.	sum of squares	mean square
Model	4	$2.45 \cdot 10^5$	$6.13 \cdot 10^4$
Error	102	775	7.60
Uncorrected total	106	$2.46 \cdot 10^5$	
Corrected total	105	$2.71 \cdot 10^4$	
Adjusted coefficient of determination $\bar{R}^2 = 99.68\%$			

Source: own creation using Mathematica.

(b) Parameter estimates

Parameter		unit	mean	std. error	95%-interval	
Start	t_0	year	2,003.44	0.182	2,003.08	2,003.81
Size	M	adopters	55.4	0.391	54.7	56.2
Innovator inflow	p	year^{-1}	0.177	0.0685	0.041	0.312
Imitator inflow	q	year^{-1}	1.135	0.173	0.792	1.478

Source: own creation using Mathematica.

5.5 Other examples

In figure 5.5 the fits are show for eight packages, the statistics are not presented. The packages and their function are:

`libnotify` is a library for notifications. In modern desktop environments applications may want to notify the user of certain events, for example a battery that is about to go empty, a new email or an incoming phone call. The adoption is relatively slow, despite its usefulness. A possible explanation is that the target applications all have their own custom solutions, which the developers are keen to keep.

`libtheora` is a library for the Schroedinger video codec. It implements a multimedia standard for use by video players. Just as with `libmad` there is a strong innovator driven growth.

`qt-core` and `qt-gui` are libraries from the Qt toolkit. These libraries provide a standardised way to, for example, open and process files or draw user interfaces. Given its rather fundamental nature the rapid growth is odd. The explanation is that the packages used to be one package, named `qt`, but got split up due to its size. It is likely that the growth represents existing dependers moving from the old to the new packages, rather than new adopters.

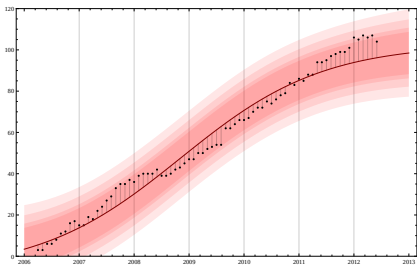
`taglib` is a library that processes metadata from multimedia files. The package allows media players to read and store information such as artist and title from multimedia files. Again, like the other multimedia packages there is rapid innovator driven growth.

`udev` is a device manager. Its task is to communicate closely with the hardware drivers in Linux kernel to monitor any changes in the hardware configuration. It represents an architectural change in a very low level component, this might explain its slow imitator driven growth.

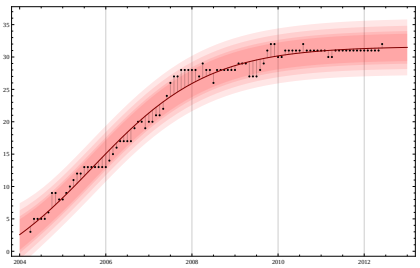
`libXaw` is a user interface library much like `qt-gui`. The project behind the package has been around for at least a decade, the extremely rapid growth is likely to be an anomaly. Perhaps the package and a collection of applications using it got added to the Portage Database around 2006.

`cairo` is a graphics library. It provides facilities for drawing lines, circles, text and other graphics primitives and is used by user graphics-heavy projects such as user interface libraries. Much like `udev` it is an architectural change at a low level, this might explain its similar growth pattern.

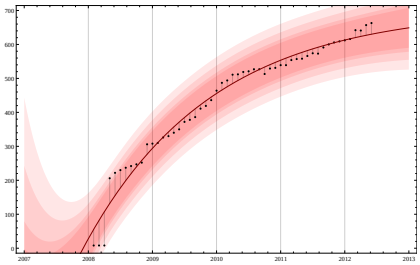
Figure 5.5: The Bass curve fitted on a number of packages



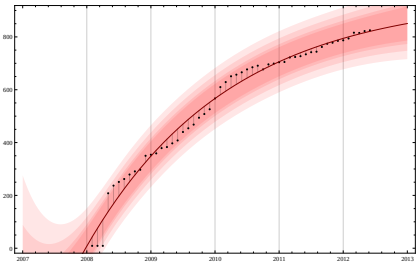
(a) `libnotify`: $p = 0.05$, $q = 0.72$



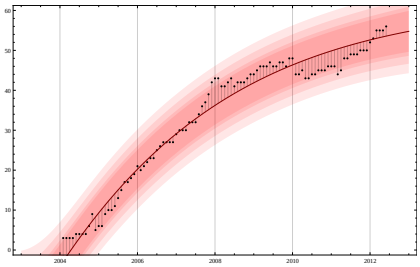
(b) `libtheora`: $p = 0.11$, $q = 0.63$



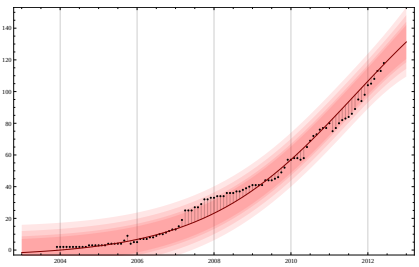
(c) `qt-core`: $p = 0.50$, $q = 0.00$



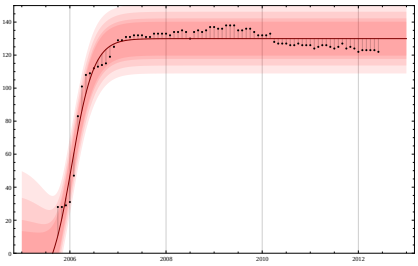
(d) `qt-gui`: $p = 0.45$, $q = 0.01$



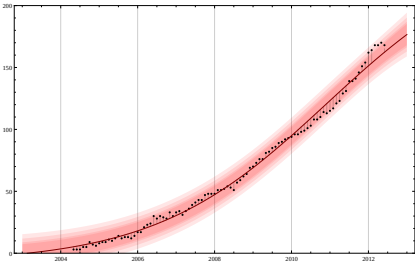
(e) `taglib`: $p = 0.22$, $q = 0.04$



(f) `udev`: $p = 0.01$, $q = 0.50$



(g) `libXaw`: $p = 0.65$, $q = 3.74$



(h) `cairo`: $p = 0.005$, $q = 0.35$

5.6 Example of growth *and* demise

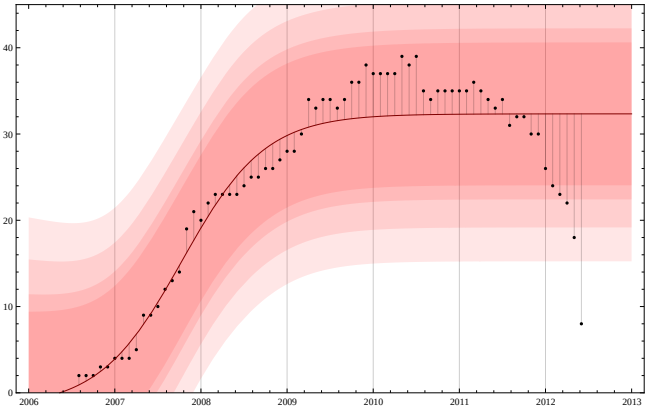
The previous examples are all about projects that start and undergo a growth phase that can be explained by a Bass diffusion process. So far, the Bass diffusion model has appeared to give a very accurate explanation of the adoption of an open source software library.

A Bass diffusion is monotonically increasing, it will always rise, but never decline. The project `libmad` (see fig 5.5b) is a good example of this behaviour. The package has an innovator driver growth that brings it close to its maximum in about two years. After that, the package's usage remains almost flat for years, and will do so indefinitely if it is a perfect Bass diffusion process. This is called the "maturity stage" in product life-cycle parlance.

A real product life-cycle will also include a "decline stage" where the product is becoming obsolete. The Bass innovation diffusion model does not account for this. In a deep sense it would not have to, once ideas spread they become part of our collective knowledge and will continue to be used by the new products being developed. But the Bass model was not developed for the spreading of ideas, it was developed in the context of marketing to model the adoption of products. Extending the Bass model to include obsolescence would be an interesting extension to venture.

The package `xulrunner` in the dataset is a nice example of a short but complete life cycle, as will soon be demonstrated. When the Bass model is applied naively and a least mean squares best-fit is made, the result is as in figure 5.6. The model fits poorly, the explained variance of 96.63% it is barely above the 95% significance mark. It is still significant that the process fits a Bass diffusion process, but only barely so.

Figure 5.6: Bass fit on the number of packages depending on the library `xulrunner`. The Bass model can explain $\bar{R}^2 = 96.84\%$ of the variance in the data.



Source: own creation using Mathematica.

When looking at the Bass model parameters in table 5.3 some parameters are still accurate. Given that the fit is not good it is striking that the introduction time t_0 and size M can still be determined with relatively small 95%-confidence intervals. From the figure it is apparent that the size is an underestimation caused by the declining values at the end. The innovator inflow and particularly

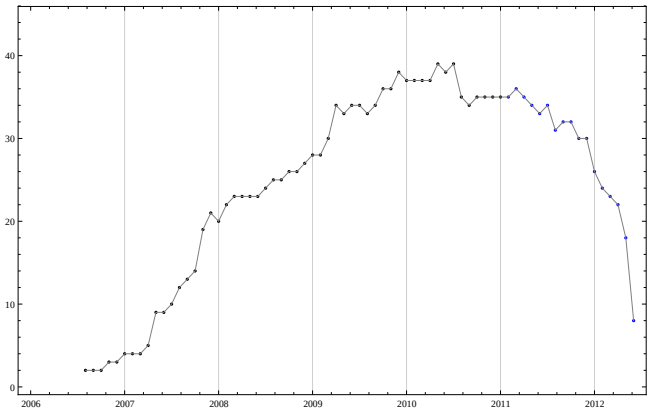
Table 5.3: Results of naively fitting the Bass model to the total number of dependers of `xulrunner`. Fitted using nonlinear least squares regression analysis. See figure 5.6 for a plot of the data and fitted model.

Parameter		unit	mean	std. error	95%-interval	
Start	t_0	year	2,006.36	0.90	2,004.56	2,008.16
Size	M	adopters	32.34	0.87	30.60	34.07
Innovator inflow	p	year^{-1}	0.10	0.22	-0.40	0.61
Imitator inflow	q	year^{-1}	1.97	0.62	-0.44	3.50

Source: own creation using Mathematica.

the imitator inflow have huge intervals.

Figure 5.7: The number of packages depending on `xulrunner` as a linearly interpolated function over time.

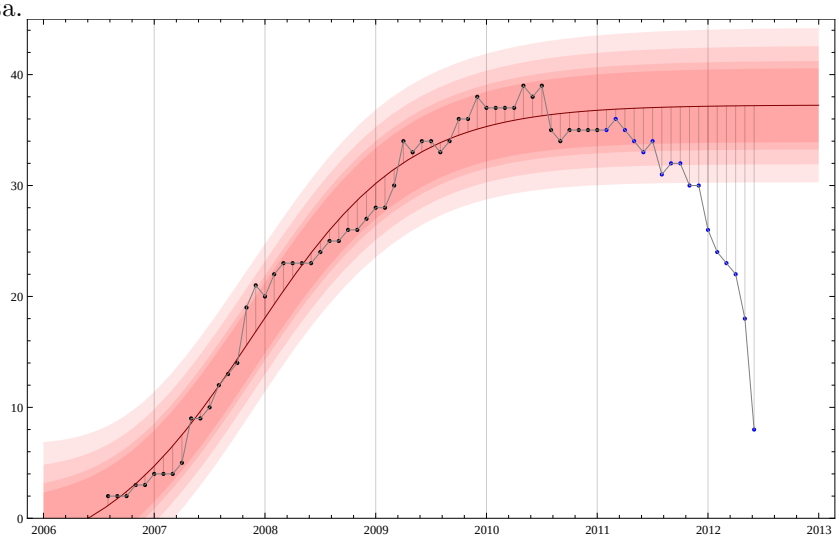


Source: own creation using Mathematica.

If one forgets the naive Bass fit from 5.6 and looks at the dependency growth on itself as in figure 5.7 the cause is clear: the package becomes obsolete, which the Bass model as presented in section 2.1.3 does not understand. The decline of the package use start from approximately 2011 onwards, the blue dots in the figure.

Excluding the blue dots, the decline from 2011 onwards, from the data results in the Bass model fit from figure 5.8. The fitness has increased from $\bar{R}^2 = 96.63\%$ to $\bar{R}^2 = 99.54\%$ and the parameters have tighter and reasonable confidence intervals. This is strong evidence that the initial adoption of the package is Bass diffusion process. To explain the last part the model should be extended with an obsolescence term. Graphs such as figure 5.7 could contain clues on how such an extension should work.

Figure 5.8: Bass fit on the number of packages depending on the library `xulrunner`. The Bass model can explain $\bar{R}^2 = 99.53\%$ of the variance in the data.



Source: own creation using Mathematica.

Table 5.4: Results of fitting the Bass model to the total number of dependers of `xulrunner`. Fitted using nonlinear least squares regression analysis. See figure 5.8 for a plot of the data and fitted model.

(a) ANOVA Table

	D.F.	sum of squares	mean square
Model	4	38,962	9,741
Error	50	170.00	3.400
Uncorrected total	54	39,132	
Corrected total	53	8,124	
Adjusted coefficient of determination $\bar{R}^2 = 99.53\%$			

Source: own creation using Mathematica.

(b) Parameter estimates

Parameter		unit	mean	std. error	95%-interval	
Start	t_0	year	2,006.40	0.20	2,005.99	2,006.78
Size	M	adopters	37.27	0.76	35.73	39.90
Innovator inflow	p	year^{-1}	0.153	0.072	0.008	0.297
Imitator inflow	q	year^{-1}	1.28	0.23	0.81	1.74

Source: own creation using Mathematica.

Chapter 6

Conclusions and discussions

In this chapter the results from the small empirical study will be reflected upon and the consequences for the broader topics from this thesis will be discussed. Specifically the viability of empirical research on open source software innovation using dependency graph analysis will be discussed. This result is then interpreted in the contexts of the intellectual property debate and innovation research in general. Finally, suggestions are given for future studies.

6.1 Conclusions from the real-world data

The dataset is large and complicated. Despite the careful selection of the dataset there was still some noise. The changes in the database format over time caused some data to become unreadable, despite considerable attempts to compensate for this. The issue with the package moves, virtual packages and meta packages where at least partially resolved, but there are still issues from package split-ups and, perhaps, mergers. In depth knowledge of both open source software development and innovation research is required to resolve these issues.

Nevertheless, vast databases containing valuable information on vast numbers of open source packages are publicly available. When specialised tools are developed to extract only the relevant information, they are still so large that, for example, cluster analysis is hard. The shear size of the datasets complicates analysis, but once these hurdles are overcome the amount insights gained in open source innovation dynamics is equally great.

Dependee graphs could be scale free. The distribution of the number of dependees per packages appears linear in a log-log plot (see figure 5.2(b)). This could indicate that the distribution is a power-law distribution, which in turn would imply that the dependee graph is scale-free, like social networks. This hypothesis was not rigorously tested, due to the unavailability of proper statistical tests.

Dependee graph cluster analysis can reveal related technologies. Graph cluster analysis proved difficult due to size of the dataset, but when it was applied to the KDE subset it did cluster related technologies.

Dependee growth is a Bass diffusion process. Overall the Bass diffusion model gave very good fits with the empirical data. Using only four parameters it was able to describe all the growth curves from the empirical data. Full statistical rigour would require a more involved analysis using the methods from, for example, Escanciano (2006), but given the amount of and quality of evidence found so far the hypothesis could be considered confirmed.

Bass parameters p and q are difficult to interpret and compare. The parameters are accurately measured, but the p and q values are difficult to interpret. A high p does not automatically mean an innovator driven growth: if the q value is also high then the result is simply a lot of growth. For the same reason it is also difficult to compare the p and q between packages. Vijay Mahajan (1995) suggests using $\frac{q}{p}$ and $q + p$, this represents the total adoption rate and an imitator/innovator ratio.

Dependee graph analysis provides new insights. The exploratory study revealed two new insights in the innovation dynamics of open source software: 1) multimedia libraries are quickly adopted by innovators and 2) low-level architectural changes happen slowly and by imitation. Further studies could test these hypotheses.

The Bass model should be extended to include discarders. The Bass model and the present analysis is formulated in terms of absolute number of users, but in most applications only sales figures are available. The number of sales is the first derivative of the Bass model, hence the model is usually applied in its derivative form Vijay Mahajan (1995). As a consequence the model only considers adopters, but does not consider *discarders*.

In the `xulrunner` example the package was being discarded from 2011 onwards, providing insights in the discarding mechanism. The next step would be to collect more examples of packages being discard, look at their patterns and develop a model of discarding to supplement the Bass model of adoption. One model could for example be the inverse of a Bass curve, this makes sense when the market share of the original package is taken over by a new package. The unique feature of dependency graph analysis to give absolute user numbers facilitates this.

6.2 Viability of dependency graph analysis

The scale and complexity of the dependency graphs and open source innovation requires some care. Three notable issues became apparent in this thesis: 1) In the open source community there is a lot of forking. It is not always clear whether a forked project constitutes the continuation of the original project or a separate new project. A more thorough study on the nature of forking could provide the insights to resolve this. 2) Due to the public nature of open source development many immature or abandoned projects are visible in the larger datasets. This is good from a scientific perspective: it allows one to research projects from their early beginning and look at projects that failed to grow or became obsolete. But it clouds the 'big picture' with many projects that

do not significantly contribute to the overall innovation. In large datasets one would have to devise a relevance metric to select the relevant metrics. Such metrics could be the number of developers, the number users or the number of dependees. 3) The shear scale of the available databases provided challenges to process. Specialist tooling was required to transform the raw data into more manageable formats.

Despite the scale and complexity a lot of information could be extracted from the real-world data. Of the methods developed in chapter 3 only the cluster analysis and the Bass diffusion model where tested, both successfully delivered new insights in open source innovation dynamics. This suggests that other methods, for example those based on the Henderson-Clark innovation patterns might also reveal interesting new insights. The dependency graph methods therefore seem a viable method of open source innovation research.

The dependency graph has an important limitation in that it does not show the end users of projects. The dependency graph only shows how projects are using each others technologies, not which technologies end users are using. While the method provides insights in the adoption of a technology presented as an independent package, such as a new media codec, it is less suitable to measure the adoption of an end user application, such as a new web browser. In the present study the projects analysed all have a sizable user base, due to the choice of dataset. The distribution maintainers need to do considerable effort to include and maintain a package in their database so they are unlikely to include packages without a moderate user base.

6.3 Implications

Implications for the intellectual property debate The motivation for this thesis is the intellectual property debate, particularly its consequences for open source software. As has been shown in chapter 1 open source software development provides an important source of innovation and strong intellectual property legislation can harm this engine of innovation. Chapter 1 showed that quantitative research to measure the effects of intellectual property legislation hardly measures open source software innovation. To resolve the intellectual property debate in the most scientific manner it is important to investigate the innovation dynamics of open source software.

In this thesis new quantitative methods where presented to analyse the innovation dynamics in open source software. These methods can provide new insights that could affect the intellectual property debate. Two examples: 1) One could try to quantify the open source innovation and reason about the potential effects of IP legislation. If it where demonstrated that the proposed IP legislation harms open source innovation more than it helps non open source development one could question the legislation's value to society. 2) If it is demonstrated that the open source method of innovation is more productive than alternative methods then one could propose to stimulate (parts of) the open source development method in different areas. (For example by recognising open source as charity for tax purposes.)

Implications for commercial software development A new method to analyse open source innovation could also contribute to commercial software

development. It is tempting to pit open source versus commercial software development and compare the two. And why not! By comparing the two best pratises could be exchanged and both could benefit. One should however be careful not to take this juxtaposition too far, because the borders are regularly crossed, as can be seen from the examples in section 1.2.

In section 1.2 it becomes clear that both small and large commercial software innovators use open source software in their innovations. This implies that open source innovation has a some positive effect on commercial innovation, it would be interesting to quantify this effect.

Implications for innovation research In this thesis it is shown that using dependency graph analysis one can study grand-scale innovation dynamics in open source software. It was found that the method allowed one to get absolute user numbers instead of sales figures, which is unique within the field of innovation research. This could provide the quantitative data to extend the Bass model with discarding. The dependency graph analysis is also unique in the shear amount and resolution of the data it provides, giving high statistical significance with much less effort than questionnaires or cases studies would require. Suitable hypotheses about innovation could be easily tested using data from the open source software community.

The implications for innovation research do not stop a what can be gained from dependency graph analysis. The study by Dahlander and Wallin (2006) used the mailing list of the Gnome developers to do social network analysis. In open source most of the technology discussions take place in public channels such as mailing lists, IRC chatrooms and fora. Usually these discussions are archived and freely retrievable, providing a detailed and honest history on why certain design decision where made. Another rich source are the public accessible management systems, these not only contain the complete source code for a project, but they contain step-by-step information on how the source code of a project developed over time and who was responsible for what part. Lastly, the larger open source projects use issue tracking software similar to commercial software developers, these systems are also publicly accessible. Issue trackers contain structured information on problems and feature request and how they are handled by the developers.

The research that can be done in open source software is unique in several respects: It is community driven innovation on a global scale, in terms of innovative output it is comparable to a whole industry of a large nation. Furthermore, almost all the development happens in the open on the internet and is archived with the archives being freely accessible. This allows research to use automated tools, like the ones developed in this thesis, to analyse global structures and dynamics. But it is also possible to go in depth and read old email conversations on why a particular strategic or engineering decision has been made.

In short: the open source software community contains a vast amount of accessible information. It is the authors opinion that the field of innovation research has only begun to scratch the surface of this resource.

6.4 Suggestions for future studies

Open source development contains a lot of project forking, a concept that is not very common in other areas of innovation. The heavy use of forking provided difficulties with defining the boundary between two developmental copies of the same project, or two diverting projects. A more thorough study on the nature of forking could provide the insights to resolve this.

The Ohloh dataset contains a lot of information on a huge amount of open source projects. If this dataset where extended with dependency information it would create a huge dataset to analyse, with even more information than the package databases provide. Give the nature of the Ohloh project it is likely that they are open to such initiatives.

A limitation of the dependency graph analysis was the unavailability of end user statistics. Such statistics could be gathered by other means: Installing a package happens by downloading the relevant files from a server. This is usually not a server operated by distribution, they have limited resources and can not afford the facilities. Therefore the distributions rely on volunteers to host copies of those files for their user to download from, so called 'mirrors'. It so happens that the Student Net Twente of the University of Twente hosts popular mirrors for several distributions, including Gentoo. They could collect download statistics for the files, which can easily be traced back to the relevant packages. Analysing these download statistics can reveal end user usage of packages.

In open source all the innovation happens publicly and is usually well archived and accessible. With proper methods these huge amounts of information can be tapped and new insights gained. Be creative!

Appendix A

Litterature

Faq on the opinion of the enlarged board of appeal, 2010. Available at `http://www.epo.org/news-issues/issues/computers/eba/faq-opinion.html`.

Paul J. Adams, Andrea Capiluppi, and Adriaan de Groot. Detecting agility of open source projects through developer engagement, 2008.

Frank M. Bass. A new product growth for model consumer durables. *Management Science*, 15(5):215–227, January 1969.

Bass's Basement Research Institute (BBRI). The bass model homepage, 2010. Available at `http://www.frankmbass.org/BassModel/`.

Knuth Blind, Jakob Edler, and Michael Friedewald. *Software Patents – Economic Impacts and Policy Implications*. Edward Elgar Publishing, Gheltenham, United Kingdom, 2005. ISBN 9781845424886.

Michele Boldrin and David K. Levine. *Against Intellectual Monopoly*. Cambridge University Press, Cambridge, United Kingdom, 2008. ISBN 9780521879286. Also available at `http://levine.sscnet.ucla.edu/general/intellectual/againstfinal.htm`.

James Boyle. *The Public Domain – Enclosing the Commons of the Mind*. Yale University Press, New Haven, Connecticut, 2008. ISBN 9780300137408. Also available at `http://www.thepublicdomain.org/download/`.

Changwoo Choi, Seungkyum Kim, and Yongtae Park. A patent-based cross impact analysis for quantitative estimation of technological impact: The case of information and communication technology. *Technological Forecasting & Social Change*, 74:1296–1314, 2007.

Linus Dahlander and Martin W. Wallin. A man on the inside: Unlocking communities as complementary assets. *Research Policy*, 35:1243–1259, 2006.

Fadi P. Deek and James A.M. McHugh. *Open Source – Technology and Policy*. Cambridge University Press, Cambridge, United Kingdom, 2008. ISBN 9780521707411.

Koen Dittrich, Geert Duysters, and Ard-Pieter de Man. Strategic repositioning by means of alliance networks: The case of ibm. *Research Policy*, 36:1496–1511, 2007.

E.C. Engelsman and A.F.J. van Raan. A patent-based cartography of technology. *Research Policy*, 23:1–26, 1994.

Convention on the Grant of European Patents (European Patent Convention, EPC), 2000. EPO: European Patent Organisation. Available at `http://www.epo.org/patents/law/legal-texts/html/epc/2000/e/contents.html`.

J. Carlos Escanciano. Goodness-of-fit tests for linear and nonlinear time series models. *Journal of the American Statistical Association*, 101(474):531–541, 2006. doi: 10.1198/016214505000001050. URL `http://www.tandfonline.com/doi/abs/10.1198/016214505000001050`.

Emmanuelle Fauchart and Eric von Hippel. Norms-based intellectual property systems: The case of french chefs. *Organization Science*, 19(2):187–201, 2008. doi: 10.1287/orsc.1070.0314.

Avery N. Goldstein. *Patent Law for Scientists and Engineers*. CRC Press, Boca Raton, Florida, 2005. ISBN 9780824723835.

Jiang He and M. Hosein Fallah. Is inventor network structure a predictor of cluster evolution? *Technological Forecasting & Social Change*, 76:91–106, 2009.

Rebecca M. Henderson and Kim B. Clark. Architectural innovation, the reconfiguration of existing product technologies and the failure of established firms. *Administrative Science Quarterly*, 35:9–30, March 1990.

Joachim Henkel and Eric von Hippel. Welfare implications of user innovation. *The Journal of Technology Transfer*, 30(1):73–87, 2004. doi: 10.1007/s10961-004-4359-6.

Sam Hovecar. Do what the fuck you want to public license, 2004. Available at `http://sam.zoy.org/wtfpl/`.

Adam B. Jaffe and Josh Lerner. *Innovation and Its Discontents – How our broken patent system is endarngering innovation and progress and what to do about it.* Princeton University Press, Princeton, New Jersey, 2007. ISBN 9780691127941.

Poul-Henning Kamp. Beerware, am i really serious?, 2004. Available at `http://people.freebsd.org/~phk/`.

Karim R. Lakhani and Eric von Hippel. How open source software works: "free" user-to-user assistance. *Research Policy*, 32(6):923–943, 2003. doi: 10.1016/S0048-7333(02)00095-1.

Hyojeong Lim and Yongtae Park. Identification of technological knowledge intermediaries. *Scientometrics*, 84:543–561, 2010.

Katherine W. McCain, June M. Verner, Gregory W. Hislop, William Evanco, and Vera Cole. The use of bibliometric and knowledge elicitation techniques to map a knowledge domain: Software engineering in the 1990s. *Scientometrics*, 65(1):131–144, 2005.

Zouhaïer M'Chirgui. Dynamics of r&d networked relationships and mergers and acquisitions in the smart card field. *Research Policy*, 38:1453–1467, 2009.

Vadake K. Narayanan. *Managing Technology and Innovation for Competitive Advantage*. Prentice Hall, Englewood Cliffs, New Jersey, 2001. ISBN 9780130305060.

A Patent Technology Monitoring Team Report – Patent Counts By Class By Year, 2010. PTMT: U.S. Patent and Trademark Office – Patent Technology Monitoring Team. Available at `http://www.uspto.gov/web/offices/ac/ido/oeip/taf/cbcby.htm`.

Joseph Alois Schumpeter. *The Theory of Economic Development*. Transaction Publishers, New Brunswick, New Jersey, 2004. ISBN 9780878556984. Translation of: Theorie der wirtschaftlichen Entwicklung (1934).

Daniel Scocco. Innovation zen: Henderson–clark model, August 2006. Available at `http://innovationzen.com/blog/2006/08/11/innovation-management-theory-part-3/`.

Cosma Shalizi. So you think you have a power law — well isn't that special?, July 2007. Available at `http://cscs.umich.edu/~crshalizi/weblog/491.html` and `http://bactra.org/weblog/491.html`.

Kirk St.Amant and Brian Still. *Handbook of Research on Open Source Software – Technological, Economic and Social Perspectives*. Information Science Reference, London, United Kingdom, 2007. ISBN 9781591409991.

The Oslo Manual. *The Measurement of Scientific and Technological Activities, Guidelines for Collecting and Interpreting Innovation Data*. Organisation for Economic Co-operation and Development (OECD) and Statistical Office of the European Communities (Eurostat), third edition, 2005. ISBN 9264013083.

D.W.F. Verkade, D.J.G. Visser, and L.D. Bruining. Ruimere octrooiëring van computerprogramma's: Technicality of revolutie? Technical Report 37, Nationaal Programma Informatie Technologie en Recht (ITeR), The Hague, Netherlands, 2000.

Frank M. Bass Vijay Mahajan, Eitan Muller. Diffusion of new products: Emperical generalizations and manegerial uses. *Marketing Science*, 14:G79–G88, 1995.

Rajendra K. Srivastava Vijay Mahajan, Eitan Muller. Determination of adopter categories by using innovation diffusion models. *Journal of Marketing Research*, 27:37–50, February 1990.

Eric von Hippel and George von Krogh. Open source software and the 'private-collective' innovation model: Issues for organization science. *Organization Science*, 14(2):209–223, 2003.