UNIVERSIDAD REY JUAN CARLOS

ESCUELA SUPERIOR DE CIENCIAS EXPERIMENTALES Y TECNOLOGÍA

EMPIRICAL SOFTWARE ENGINEERING RESEARCH ON LIBRE SOFTWARE: DATA SOURCES, METHODOLOGIES AND RESULTS

DOCTORAL THESIS

Gregorio Robles

Ingeniero de Telecomunicación

Madrid, 2005

Thesis submitted to the Departamento de Informática, Estadística y Telemática
in partial fulfillment of the requirements for the degree of

Doctor europeus of Philosophy in Computer Science

Escuela Superior de Ciencias Experimentales y Tecnología
Universidad Rey Juan Carlos
Móstoles, Madrid, Spain

DOCTORAL THESIS

EMPIRICAL SOFTWARE ENGINEERING RESEARCH ON LIBRE SOFTWARE: DATA SOURCES, METHODOLOGIES AND RESULTS

Author:
Gregorio Robles-Martínez
Ingeniero de Telecomunicación - Telecommunication Engineer

Director:
Jesús M. González-Barahona
Doctor Ingenierio de Telecomunicación - Doctor Telecommunication Engineer

Móstoles (Madrid), Spain, 2005

DOCTORAL THESIS:	Empirical Software Engineering Research on Libre Software: Data Sources, Methodologies and Results
AUTHOR:	Gregorio Robles-Martínez
DIRECTOR:	Jesús M. González-Barahona

The committee named to evaluate the Thesis above indicated, made up of the following doctors:

PRESIDENT:	Prof. Dr. Manuel Hermenegildo (Universidad Politécnica de Madrid, Madrid, Spain)
VOCALS:	Prof. Dr. Brian Fitzgerald (University of Limerick, Limerick, Ireland)
	Dr. Stefan Koch (Wirtschaftsuniversität Wien, Vienna, Austria)
	Dr. Daniel M. Germán (University of Victoria, Victoria, Canada)
SECRETARY:	Dr. Antonio Fernández-Anta (Universidad Rey Juan Carlos, Madrid, Spain)

has decided to grant the qualification of

Móstoles (Madrid, Spain), February 9th 2006

The secretary of the committee.

To my father, mother, sister and Nuria
for their infinite love, understanding and support.

Acknowledgments

It has been much work during these last four years. But at the same time, it has been a lot of fun. A great part of this fun is due to the people I've been involved with, both in my research and teaching duties and in my personal life (although sometimes I don't know where the former ends and the latter starts).

I have to start showing my gratitude to Jesús, who has officially been my advisor, but unofficially much more. There has been a special connection between us from the first minute regarding what we wanted to research and how we wanted to do it. With much effort and many nights without sleep (which I myself can easily afford, but which I should regret and apologize in front of Jesús' family) we have started to build a new research line, many research projects and a new research group that accounts now up to 8 persons. My name is the only one that appears on the front cover of this thesis, but without any doubt it would be fair if the name of both of us would be there. I won't have time in this life to say enough times *Muchas gracias, Jesús.*

Of course, I have to mention all the guys from our research team (Juanjo, Israel, Teo, Álvaro N., Jorge, Diego and Carlos, as well as Luis L. and Álvaro del C.). They have brought new ideas and lots of amusement into my research and I'm sure they all will agree when I say that we've built a small family in our labs in Móstoles. I'm also very thankful to all those who compose the Grupo de Comunicaciones y Sistemas (GSyC) and are not part of the CALIBRE team: José (with whom I share the pride of being a Real Madrid supporter), José María (when my friends tell me that I work too much I always explain them that I've got a colleague named José María...), Agustín, Pedro, Vicente, Luis R., Miguel, Eva, Paco, Pablo, Rafaela, Sergio, Antonio, *nemo*, Gorka, Katia, Andrés, Juan and Quique.

During these years of research, I've had also the possibility of visiting some research groups abroad; in 2003 I visited Vienna for 8 weeks and in 2004 I was in Maastricht for four months. The time I spent there has had a great impact on this work, but I also enjoyed much the people I found there. So, many thanks go to Stefan (plus Lale and child), who acted as local host in Vienna, and to Rishab in Maastricht, also to the many great people I had the possibility to know there: Rüdiger, Bernhard, Wilma, Ekin, Semih, Abraham, Paolo, Elad, among others. I cannot forget all those with whom I have co-authored some of the works or actively exchange some research ideas in these last years. In this sense, Martin Michlmayr (from the Univ. Cambridge), Andrea Capiluppi (Univ. Torino and Open Univ.), Juan F. Ramil (Open Univ.), Ioannis Samoladas (Univ. Thessaloniki), Juan Julián Merelo (Univ. Granada), Paul A. David (Univ. Stanford & Univ. Oxford) and Jorge Ferrer (Univ. Politécnica of Madrid) will be somewhat *guilty* of some of the ideas and concepts that are included in this thesis. Rosario Plaza should have her place here as well, as she has been the one who has reviewed my English with much patience and dedication.

I also have to mention all those who have been the *victims* of this PhD thesis: my friends. They have been the ones that have suffered the lack of time I had to devote them and that they, of course, deserve. Many thanks go to Diego, Rodrigo & Cristina, Carlos Martín Ugalde, Enrique Zamora, José María Nadal, my scouting group, the people of Scouts-es, some of my students, and many, many others...

Finally, I will always be in debt with my family. My father, my mother and my sister Elisa have always been a great support and have had the necessary understanding and patience that is required when you are dealing with a PhD student at home. A special hug goes to my grand parents, Gregorio and Celia, and Ramón and Maria del Carmen. And last but not least, I'm deeply grateful to Nuria,

who has been always there showing infinite love.

Gregorio
Madrid, October 2005

Abstract

With the appearance and implantation of Internet new ways of developing software have arisen that make use of telematic tools, follow flexible methodologies and incorporate third-party contributions. One of the paradigmatic examples of software development that counts on the aforementioned characteristics can be found in the phenomenon of libre (free/open source) software, being of special special interest those projects that are large in number of participants and in software size.

Although at first these new environments are less controllable than traditional ones (because development is done generally in a geographically distributed way, there is no a company behind the development that takes the lead, traditional hierarchic structures are not followed or external contributions are hardly predictable), we have access to much information: the software product itself and many of the by-products that are created during the development process (communication archives, bug-tracking systems and versioning systems, among others). These data sources are usually publicly available on the Internet, so we can make exhaustive analysis with a great amount of data (much of which is hardly obtainable in traditional, industrial environments).

The goal of this thesis is to identify the data sources that libre software projects offer publicly, to present and display some methodologies for the analysis of these sources and the data that we can extract from them, and to show the results that have been obtained from applying these methodologies. Our intention is, in particular, to know the libre software phenomenon better, but also in general software creation processes since the acquired knowledge does not have to be specific to libre software, but could be applied to many other development environments.

Thus, we will start in this thesis with the description of the publicly available data sources on the Internet and the data that we can extract from them. Afterwards, several methods, that will depend on the source, will be used to obtain information from the data and to filter out *interferences*. Finally, several methodologies will be presented and applied on the data obtained from libre software projects which have been selected as case studies. The methodologies will range from *classical* to novel ones. Thus, among the *classical* we will perform an analysis of the growth of the software systems as it is known from software evolution, or we will apply social network analysis, a technique from the field of social sciences. In both cases, the contribution of this thesis has been to apply them to libre software projects. Regarding novel methodologies, we propose the *archaeological* analysis of software systems with the aim of stating what remains from previous versions, the generalization of software evolution to file types different from source code (for instance, documentation, translation or user interface files, among others) or the study of the evolution of volunteer participation and the regeneration of the leading "core" group. Also, a series of tools have been created to automate, at least partially, the whole process. These tools permits to reuse these methodologies on other projects.

Among the main contributions of this thesis we can state that this is the first exhaustive analysis of a large number of software projects, although the proposed methodologies and the tools that have been developed allow the study in the next future of more projects. On the other hand, we have shown that the technical analysis should be complemented with socio-technical analysis to fully understand the development process and many of the technical issues of (libre) software projects.

Resumen[1]

Con la aparición e implantación de Internet han surgido nuevas maneras de desarrollar software que hacen uso de herramientas telemáticas, emplean metodologías flexibles e incorporan contribuciones por parte de personas externas al equipo de desarrollo. Uno de los ejemplos paradigmáticos de desarrollo software que cuenta con las características mencionadas se puede encontrar en el fenómeno del software libre, siendo de especial especial interés en proyectos de gran tamaño (en número de participantes y en tamaño del software).

A pesar de que en un principio estos nuevos entornos son menos controlables que los tradicionales (debido a que el desarrollo se realiza generalmente de manera distribuida geográficamente, a que no hay detrás una empresa que lleve la mayor parte del desarrollo o, al menos, no se sigan las estructuras jerárquicas tradicionales y a que las contribuciones externas son difícilmente predecibles), gracias a que tanto el producto del desarrollo (el software en sí) como muchos de los subproductos que se producen durante el desarrollo (trazas de las comunicaciones utilizadas para la comunicación del equipo de desarrollo, sistemas para almacenar los errores del software o sistemas de control de versiones) se encuentran normalmente disponibles de manera pública en Internet, podemos realizar análisis exhaustivos con gran cantidad de datos, muchos de ellos difícilmente conseguibles en ámbitos tradicionales.

Esta tesis se tiene como objetivo identificar las fuentes de datos que ofrecen los proyectos de software libre de manera pública, presentar algunas metodologías para el análisis de las fuentes y de los datos que podemos extraer de las mismas, y mostrar algunos resultados de las metodologías empleadas para proyectos de software libre. Se pretende con ello conocer mejor el fenómeno del software libre, en particular, y los procesos de creación de software, en general, ya que parte del conocimiento adquirido no tiene por qué ser específico del software libre, sino que puede aplicarse en cualquier otro entorno de desarrollo.

El procedimiento que se seguirá en esta tesis partirá de la descripción de las fuentes de datos, así como de los datos que podemos extraer de las mismas. Posteriormente, se presentarán varios métodos, dependientes de las fuentes, para la obtención de información a partir de los datos, así como el filtrado de *interferencias*. Finalmente, se presentan varias metodologías que se aplicarán sobre los datos de proyectos de software libre que han sido elegidos como casos de estudio. Estas metologías serán tanto *clásicas* como novedosas. Así, entre las *clásicas* podemos nombrar el análisis del crecimiento de software tal y como se conoce en evolución de software o el análisis de redes sociales, del campo de las ciencias sociales. En ambos casos, la contribución de esta tesis ha sido su aplicación a proyectos de software libre. En cuanto a metodologías novedosas, se propone el estudio *arqueológico* del software con la finalidad de constatar qué es lo que preservan las versiones actuales del pasado, la inclusión de artefactos fuente diferentes al código fuente escrito en un lenguaje de programación, como pudieran ser archivos de traducción, documentación o interfaz de usuario o el estudio de la evolución de la participación y regeneración del equipo de desarrollo. Asimismo, se han creado una serie de herramientas que automatizan, al menos parcialmente, todo el proceso lo que permite la reutilización en otros proyectos software.

Entre las principales contribuciones de esta tesis podemos constatar que se trata del primer análisis exhaustivo de un gran número de proyectos software, aunque las metodologías propuestas y las herramientas que se han desarrollado para tal efecto permitan en un futuro próximo el estudio de todavía

[1]En el apéndice C se puede encontrar un resumen suficiente en castellano que cumple con los requisitos del artículo 24 del capítulo V de la "Normativa para la Admisión del Proyecto de Tesis y Presentación de la Tesis Doctoral" para las tesis que sean presentadas en otros idiomas diferentes del español, como es el caso de ésta.

más proyectos. Asimismo, se ha podido comprobar la importancia que tiene en la era de Internet complementar los análisis técnicos realizados sobre el producto con estudios socio-técnicos de las personas que están detrás del desarrollo del software.

Contents

List of Figures

List of Tables

Chapter 1

Motivation

Computers are useless. They can only give you answers.
Pablo Picasso

The topic of this thesis is the study, from a software engineering perspective, of the development information that is publicly available. It specifically deals with information obtained from repositories of libre software projects. This is interesting because of (at least) two reasons. On the one hand, libre software projects provide -probably for the first time in the history of software engineering- huge quantities of publicly available data. In this type of environments, source code and many other byproducts of the development process are available. Consequently, we can extract information from the source code and from traces in the main communication channels (mailing lists, forums, and others) or from development-supporting tools (bug-tracking systems, versioning systems, etc.) among others. On the other hand, the way libre software projects are developed has proven to be successful for, at least, an ample number of them. Processes and practices are based on software engineering principles which have been uncommon in the software industry during the last decades. Among these practices we can mention those of global software development, peer reviewing, rapid prototyping, self-organization or lowering the barrier for user contributions.

This thesis is therefore devoted to show the first analyses and results from the publicly available information (presented as a first reason) in order to understand processes involved in the development of libre software (which is the second reason). The goal in the long run is to have a means by which we could analyze (libre) software projects quantitatively in a semi-automatic manner. Due to the intrinsic complexity of the software processes associated to the development of libre software, we will see that a multi-facet analysis, with methods and techniques from various fields, is required.

The remainder of this motivational chapter is devoted to a brief introduction of the disciplines and fields where this work may be categorized. The key fields and concepts that this thesis touches are shown in figure 1.1, noting that they have been applied to libre software projects.

At first, we will see what libre software is and will discuss some concepts that have recently arosen in literature regarding its development model. We will present there why we specifically interested in libre software and comment on the type of libre software projects we target at in our analyses. In any case, and although the analyses and techniques presented in this dissertation have been designed with the intention of analyzing libre software projects, many of them may be applicable to non-libre software projects.

The main discipline where this work may be classified is into the software engineering field of study. We will see why libre software provides software engineering with a new perspective given the amount of publicly available data that researchers may obtain. The analyses will hence have a strong empirical component, being metrics and other quantitative methods of great importance for the goals of this work. Especially interesting are also all issues related to mining software repositories; although the data is publicly available, we will have to face some problems to obtain it in a meaningful and complete manner. Efforts and methods for doing this, in a general and almost automatic way, will be shown.

The software repositories we are going to mine usually offer longitudinal data which provide tremendous opportunities for studying of the evolution of projects. We will hence introduce “classical”

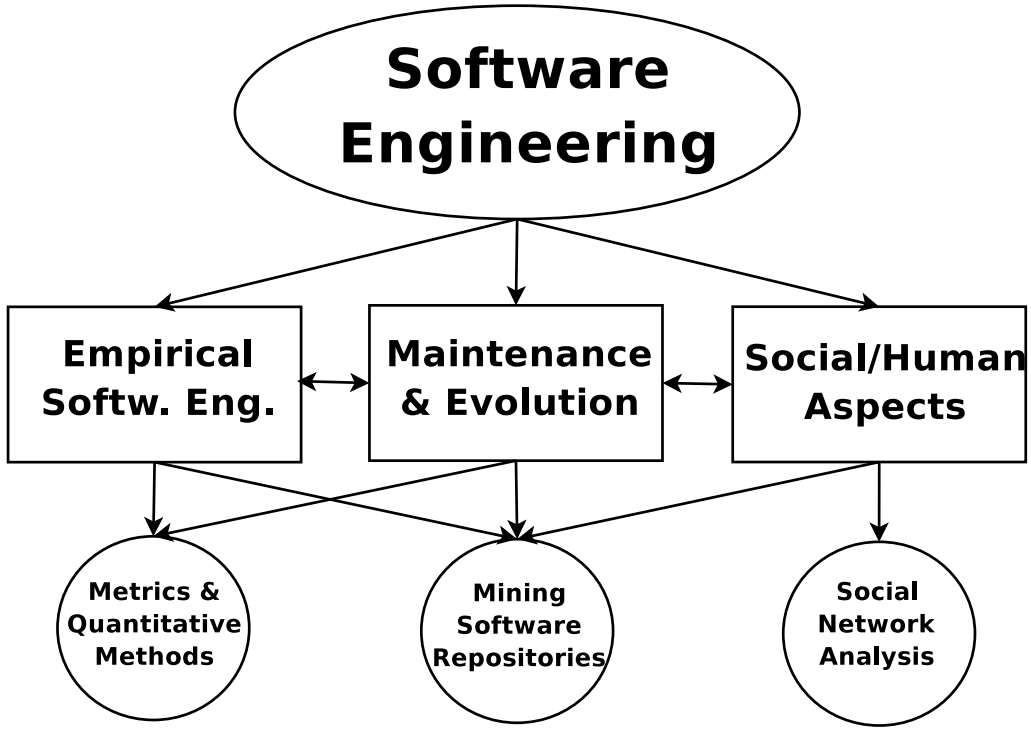


Figure 1.1: *Key fields, concepts and techniques (and their relationships) dealt with in this dissertation. As a field of study, the libre software phenomenon and especially large libre software projects (in software and community size) have been selected.*

software evolution studies, and analyze how we can use the data for other purposes, such as to specify the maintainability of the software.

Beyond technical questions, in this dissertation we will study the human and social aspects related to the development of software. These are especially important in libre software environments as large projects are heavily based on the integration of new members into the development team and on fostering external contributions made by volunteers. Our intention is to go beyond social network analysis and to try to find out how communities around large libre software projects are, how they are structured, and how they change over time, among other issues.

1.1 Development of libre software

Libre software has lately attracted, without any doubt, a lot of attention in academia, industry and among end-users. Although its philosophical principles go back to the eighties[1], it has not been until a few years ago when with the proliferation of Internet connections it has shown itself as a valid development and distribution model. There is an ample debate about the implications and future possibilities that libre software offers. In this regard, many argue that it is becoming a threat for the traditional software model, where *closed* development environments and business practices oriented towards selling software licenses are the rule.

The libre software phenomenon has supposed a major challenge in the way software is created. New paradigms regarding legal, technical, knowledge dissemination and governance issues have converted it into an interesting research topic. In the case of software engineering, the set of technical and technological practices in some large libre software projects has awoken enormous interest. But, even though the most prominent elements for software engineering researchers have been technical ones, in the author's opinion, other aspects related to libre software cannot be left aside: concepts like *community* and *social interaction* are of great importance when interpreting and understanding many

[1] In fact, before the eighties libre software already existed *de facto*, being even its history as old as that of the own software [Salus, 2005; Williams, 2002].

questions.

1.1.1 Definition of libre software

Libre software is software that has been released under the terms of a license that complies with the four freedoms [Stallman, 1999; Stallman *et al.*, 2002]. These freedoms are:

- The freedom to run the program, for any purpose.
- The freedom to modify the program to suit your needs. (To make this freedom effective in practice, you must have access to the source code, since making changes in a program without having the source code is exceedingly difficult.)
- The freedom to redistribute copies, either gratis or for a fee.
- The freedom to distribute modified versions of the program, so that the community can benefit from your improvements.

The definition of libre software does not explicitly mention the requirement of having the possibility of accessing the source code, but the freedoms to modify, adapt and redistribute the software implicitly point out that the source code can be obtained from the authors. Besides, there is no mention of the development process in the definition. The freedoms that make a software libre are related to the way a person can use, copy, modify and redistribute a software, not to the methods used for its development.

We will develop the dichotomy between the freedoms and the development methods below (in subsection 1.1.2), but before we should explain the rationale for the use of the term "libre software" in this dissertation. Because of the polysemic meaning of *free* in the English language, some notables of the libre software phenomenon launched in 1998 the Open Source Initiative and created the Open Source Definition [Perens, 1999]. Besides avoiding the confusion with the term *free*, they aimed to promote the technical advantages of the availability of source code. Although the set of licenses that conform with the Open Source Definition is with some minor exceptions the same as the one considered free software by the Free Software Foundation (the main proponent of the term "free software"), this initiative has been strongly criticized by the latter arguing that freedom is placed into a secondary position with the term Open Source.

In this thesis we will use the term "libre software" to refer to code that is distributed under a license that conforms to either the definition of "free software" (according to the Free Software Foundation) or of 'open source software" (according to the Open Source Initiative). Hence, licensing issues will not be considered in this work.

The roots of the term libre software can be found in Europe (*libre* is used in Spanish and French and is very similar to *livre* in Portuguese, to *libero* in Italian and to the word used in other Romance languages) and this term a proposal to have a definite term which unambiguously denotes freedom and at the same time can not be mistaken with a zero price. Although the use of the term "libre software" is (still) unusual in English-speaking environments[2], some important institutions (such as the European Commission) and EU-funded projects (such as the FLOSS project[3] and its continuators FLOSSPols[4] and FLOSSWorld[5] or the CALIBRE coordinated action[6]) use it.

1.1.2 Libre software, the cathedral and the bazaar

Libre software has mainly been a matter of software engineering research because of its development methods and processes. To counter the existing general idea that there is a unique method to create libre software, we must remark that this is not true: any project uses (or may use) its own model.

[2]For further historical information about the term libre software can be found in an article at the following URL: http://sinetgy.org/jgb/articulos/libre-software-origin/.

[3]Free/Libre/Open Source Software: Survey and Study (acronym FLOSS): http://www.flossproject.org.

[4]Free/Libre/Open Source Software: Policy Support (acronym FLOSSPols): http://flosspols.org.

[5]Free/Libre/Open Source Software: Worldwide impact study (acronym FLOSSWorld): http://www.flossworld.org.

[6]Co-ordinated Action on LIBRE software (acronym CALIBRE): http://www.calibre.ie.

The main reason for this is, as we have already seen, that the definition of libre software does not specify any technical indications on how software should be created. But, although any project could develop the software on its own way, the experience of the last twenty years has shown us that projects following certain practices have a higher tendency to be successful[7]. And the result is not a static picture; new practices and methods are continuously tested and introduced in order to make the development process more efficient. A paradigmatic example of a new practice can be found in the recent use of bug-tracking systems, which were introduced in the late nineties and are now commonly used in large libre software projects.

So even if any libre project could follow its own development methodologies and practices, some common characteristics have been identified in those projects which have achieved a large number of contributors. In a seminal work by Eric S. Raymond labeled "The Cathedral and the Bazaar" [Raymond, 1998], the author established a parallelism on one hand between the way *cathedrals* were built in the middle ages and how software has traditionally been build and on the other between the functioning of an oriental *bazaar* and the way some libre software projects work.

For Raymond, building cathedrals required fixed roles (architects and workers), tight rules, no interaction from the outside and a defined hierarchical decision structure. Software projects developed using "traditional" methodologies coincide in defining roles (project managers and programmers), in having central planning and in management and providing low interactivity with end-users. In opposition to this, Raymond situated the bazaar. In a bazaar, exchanges occur spontaneously and are not directed by anybody. In addition, there are no fixed roles and if there are, they change frequently. Using the analogy in the software development field, the *bazaar* is a way of developing software that aims to open the development process as much as possible, offering the possibility to participate in development and distribution activities to anyone. As a case example for such type of projects, Raymond presented some experiences from the Linux kernel and from fetchmail, a smaller project led by himself.

There exists an widespread confusion that we should clarify as soon as possible: libre software and the bazaar model are not synonymous. Freedom in software is a legal aspect that derives from the Free Software Definition [Stallman, 1999] and is reflected in a software license. The *bazaar* is a concept that belongs to the technical scope. To emphasize this point, we should mention that Raymond cited well-known libre software projects such as GNU Emacs and the GNU Compiler Collection (GCC) as projects that had a *cathedral* model although they were libre software. Actually, as Healy et al. have demonstrated, a large amount of libre software projects are composed of a single developer or a small development team where a *bazaar* development model is not possible [Healy & Schussman, 2003].

Still, a precise definition of the bazaar does not exist. There is no set of conditions that a project has to obey or accomplish to be considered as developed following the *bazaar* model, so we cannot know what projects belong to the *bazaar* and which do not. To add more controversy to this issue, there are even some authors that argue that software development forms that are widely considered as *cathedral*, for instance Microsoft programs, are really built by a *bazaar* [Bezroukov, 1997]. Although an exact definition does not exist, we could say that there exists a consensus on the fact that all these projects belong to such a development model.

For the purpose of this dissertation, we will focus on projects which are large in software size and in user community. The rationale for this is that these types of projects have aroused most interest in the software engineering community. Although large projects constitute a minority among the more than 100,000 calculated libre software projects that are available today, they have attracted the attention mostly of the public, the industry and the academia. From the software engineering perspective, they have a set of characteristics that makes them an interesting matter of study: they have an open development model to occasional contributors; they offer external contributors the possibility to get integrated into the development team; there are large amounts of contributions made by volunteers; they are developed in a distributed (global) software development environment; they foster frequent interactions with end-users; they make heavy use of telematic tools; and they offer flexible and dynamic

[7]It should be noted that there is no clear definition of what a successful libre software project is. In traditional, industrial environments success is generally measured in economic terms; i.e. a software project that provides benefits. For libre software, other reasons should obviously be considered. For a detailed discussion about defining success in the libre software world see [Crowston *et al.*, 2003b].

processes and management models. Some authors have argued that the libre software phenomenon does not add any innovative ideas to the software engineering landscape [Fuggetta, 2003]. Even if the elements cited above do certainly appear in other ways of software development, this set of characteristics can only be found as a whole in libre software. So, if maybe not all elements are exclusive to libre software, the sum of all of them is unique and worth to be considered as a matter of study.

1.2 Software Engineering regarding Libre Software

In a famous essay written in the mid eighties entitled "No silver bullet", Frederick P. Brooks stated that there is no perspective in the near future that will make software engineering improve an order of magnitude [Brooks, 1987]. Nonetheless, he proposed a set of aspects which he thought could push the development if not an order of magnitude at least the most promising achievements:

- Buy versus build. It is better to obtain components from third parties than to create them by ourselves. The explanation that usually is given for this recommendation is that project managers and developers tend to minimize the amount of work that has to be devoted to build a new software system from scratch and to maximize the effort required to (re)use an already existing system.

- Requirements refinement and rapid prototyping. Allow to know with more detail what is wanted as feedback is augmented.

- Great designers. Errors found in early stages of the development process are easier (and thus cheaper) to resolve, especially attending to the architectural design of the software. Having good designers with much experience in the team makes this more probable.

Some authors have argued that all these points are present in the development of libre software [Daffara, 2002]. The first recommendation can be transformed into taking already existing code or using libraries or components from other libre software projects. By doing this, we should have code that has already been tested and debugged by a third party.

The second point is widely known as the "release early, release often" paradigm in the libre software world (as it has been coined by Raymond [Raymond, 1998]). Following the way Linux and other prominent projects have evolved, a small and promising prototype should be published at the beginning of the project and subsequent releases should be made periodically. In projects with an ample surrounding community, feedback (suggestions and error reports) from testers and end-users is obtained and can be used both to enhance the quality of the software and to lead the future development.

Finally, to have great designers is indirectly due to meritocracy. As we will see in the related research (chapter 2), an extensive body of literature demonstrate that a small amount of very active developers is responsible for a vast portion of the total code and activity that concerns libre software projects [Ghosh & Prakash, 2000; Mockus *et al.*, 2002; Koch & Schneider, 2002; Dinh-Trong & Bieman, 2005]. These developers are known as the *core group* and serve generally as leaders of the project. Becoming part of this group requires recognition by the peers and this supposes an important contribution to the project. So, in general, the core group is composed of members who are experts and skilled in the technologies and in the project domain.

But beyond Brooks' (qualitative) recommendations and, independently of how well or badly libre software may fit them, there is an additional (quantitative) characteristic that should be considered and which is a central to this thesis: the measurement of the software development process. In this sense, libre software constitutes a very good benchmark for comparing software practices and techniques, and it is an excellent framework for studying software engineering. Measuring the development process may allow managing the project, both from the technical point of view or from the social perspective. This is especially significant in libre software environments where feedback has shown to be of great importance and where the integration of external members constitutes one of the basic elements for a

sustainable development. Thus to have up-to-date information over the project's life-cycle will enable developers and managers to be on top of it.

1.2.1 Empirical Software Engineering

Empirical software engineering is a field of software engineering aimed at applying empirical theories and methods for understanding and improving the software development process and product. Many studies in software engineering have historically suffered from a lack of data. Even if traditionally software environments have produced this kind of data, they have not been accessible to researchers. In the seldom cases where data was accessible, it could not be offered publicly to other researchers that could verify and reproduce methodology and results.

Libre software provides software engineering researchers not only with the possibility of accessing to a large amount of data that is publicly available on the Internet. Anyone can gather it in a (almost) non-intrusive way, so reproducing studies and performing new analyses on the same data set is possible. This situation has not been common in software engineering practice during the last decades, although in recent times some efforts towards sharing datasets publicly have been started (for example, the PROMISE Software Engineering Repository [Sayyad Shirabad & Menzies, 2005]).

Mining Software Repositories and quantitative methods

An important part of this work will be devoted to describe the data sources, in general software repositories, that will make our analyses possible. Software repositories go beyond the product of the development process (the software itself) and include source control systems, archived communications between the developers, bug tracking systems, and the archives of other telematic tools that are used during the development process. In the case of libre software, the availability of software repositories is, in fact, a consequence of its distributed and open nature. All this information is made available so that developers all around the globe can participate in the development process. It is also public in order to increase the diffusion of information among project participants and others who may want to collaborate in the future.

There is a voluminous literature devoted to mining software repositories in recent times, both in the proprietary and in the libre software realms. Closely related to these issues are the works that have focused on historic databases on software development, extracting information from them to better understand and improve the underlying development process [Mockus & Votta, 2000; Gall *et al.*, 1997; Atkins *et al.*, 1999; 2002; Graves & Mockus, 1998]. Some authors have looked at the versioning system, reporting the impact of software tools on the development process [Atkins *et al.*, 1999; 2002] or at the identification of the reasons for changing the software [Mockus & Votta, 2000; Zimmermann *et al.*, 2005].

From the research perspective, there are many potential benefits from mining software repositories as mining allows to improve software design and reuse in order to gain information that supports software maintenance and evolution and to track methods and practices in use. And all this information can be obtained empirically and shared freely (considering at the same time privacy issues for data related to developers, as we will see in section 3.7). Some efforts towards automatic measurement of information surrounding these software projects have been proposed [Germán & Mockus, 2003] and much of the effort of this thesis has precisely been concentrated on automating data retrieval, extraction and analysis.

One of the goals of software engineering has been to measure software projects and to have a small set of parameters that allow characterizing a software project. The amount of data that we handle and the various perspectives that we take into account in the research of libre software projects makes the use of metrics and indexes essential.

Metrics characterize a special attribute of an entity by assigning a number (in some cases, a symbol) to this entity [Fenton, 1991]. Fenton differentiates between three types of entities (processes, products and resources) and also differentiates for each entity between two attributes that can be measured: internal and external. Internal attributes can be measured on the entity, while external attributes have to be set in a context in order to be measured. Processes are activities that usually involve having

time as a factor. Internal attributes for processes are time, effort or the frequency of a factor arising. Products are given by the source code or the documentation produced throughout the whole life-cycle. Resources are the inputs used for the development and include, among others, human resources and working conditions.

In this thesis we will focus generally on some internal attributes of these three entities. The time axes will have special consideration regarding processes (with data extracted from source control systems, communication archives and defect tracking systems). Product metrics will be considered in the analysis of software size and software growth, while there will be many measures focused on human resources, i.e. developers and other contributors working on libre software projects.

1.2.2 Software Maintenance and Evolution

Software maintenance is the last phase of the software life-cycle, which starts with its first release and has to face the problems of correcting errors and enhancing the software to adapt it to new needs arisen. The definition of software maintenance by the IEEE standards is "[t]he modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment".

Although historically the software development process previous to the first release has been the one that has received most of the attention and resources, software maintenance is the software engineering activity that takes most of the life cycle of a software system. According to some studies, this is the most costly phase, ranging from one to over one hundred times the development cost [Sayyad Shirabad, 2004; Boehm, 1975; Schach, 2002].

The fact that libre software is prototypic means that from a very early stage of its development process software maintenance (and evolution, see below) takes place. There have been defined several types of software maintenance: corrective, perfective and adaptative [Swanson, 1976]. Corrective maintenance is the removal of errors, perfective maintenance is performed to enhance the software system and adaptative maintenance is carried out when the software needs to be adjusted to a new hardware environments.

Strongly tied to software maintenance is the concept of software evolution. The term software evolution does not have a standard definition, although it has been commonly adopted by industry and academia because of its growing importance in recent years. Arthur describes software evolution as "[a] continuous change from a lesser, simpler, or worse state to a higher or better state [for a software system]" [Arthur, 1988]. The most cited works in this field are those by Lehman et al., who have promulgated over the years a set of *laws* of software evolution empirically observed in industrial software systems [Lehman & Belady, 1985; Lehman *et al.*, 1997; Ramil & Lehman, 2000b; Lehman & Ramil, 2001; Lehman *et al.*, 2001]. In recent years, there have been some efforts (which will be presented in chapter 2) have been devoted to investigate if these *laws* are applicable to the development of libre software.

Both the facts that software evolution is by tradition empirically-oriented and that it looks at how data changes over time make this body of research to be very close to what is the core of this thesis. From the software maintenance point of view, we will present some techniques that will permit learning if the software has been maintained or not. We will try to infer from this information how maintainable the project is. Regarding software evolution, we have entered the debate about libre software projects, fulfilling or not the *laws* of software evolution or not, by studying a set of libre software applications. In addition, we have also studied the evolution of a software compilation (i.e. the Debian GNU/Linux software distribution) over time.

1.2.3 Social aspects of (libre) software development

The final theme of this thesis is dedicated to those who are behind the development of libre software: the developers. This study is made possible empirically, because we are able to track the individual activity of participants in the software repositories. Beyond any doubt, the management of human resources is an important part of all software developments. In the case of libre software, we are frequently confronted with activities that are done by volunteers. This has to be seen not only at the

individual (developer) level, but also in an aggregated manner. Concepts like *community* or *feedback* have gained a lot of importance and discriminate those projects being successful from those projects which will have no more than a dozen end-users.

Taking this into account, it sounds reasonable to study in depth how developers relate with among themselves, how they are organized, how they take decisions, and how they distribute the work load, among other questions. The application of techniques already used in other fields, such as social network analyses, may help to identify those developers who have strategic positions in the community. Considering the distribution of work, we will study if there exists specialization in the community, i.e., if some developers are keener to work on certain aspects of the project while others center themselves on other aspects.

On the other hand, it is interesting to note the question of the evolution of the community along a period of time. We have already shown that some works have found that a small set of persons is responsible for a high amount of activity in libre software projects. We will try to find out how the composition of this group evolves, and if it evolves at all, how the integration process of new members takes place.

All these activities are a first step towards advancing in the research of the productivity of single developers in libre software, as well as learning how productivity is affected by the *environment* in libre software projects. These questions are especially complex in the libre software world in comparison to *traditional* software developments as most developers are volunteers and here task-assignment does not follow the same rules.

1.3 Goals of the thesis

The main goal of this thesis is to gain some understanding over the processes and factors that are important in the development of libre software projects. We will therefore use a software engineering point of view and benefit from the large amount of data that such type of projects provide publicly over the Internet. In other words, this work will try to obtain a better understanding of the libre software phenomenon analyzing methodically and empirically publicly available traces from the software development process.

This thesis should be classified among the efforts carried out for a better understanding of the libre software phenomenon. Because of its recent popularity and of some of its characteristics (distributed development, involvement of volunteers, self-organization, among others), libre software has to deal with more complexity than *traditional* software development environments. The fact that it has been only partially understood has provoked that a certain mystification of this phenomenon exists. Many personal opinions and perceptions in the first studies together with a lack of data helped to build up this idea. Our approach in this dissertation is hence closer to data and facts.

To a certain point, we may find in medical exams a good analogy of the goals of this dissertation. In the medical sciences, a doctor may *read* from the data of a blood analysis how healthy a patient is and which values should be corrected. Our aim in the long run with this thesis is to have a similar approach for software projects (as depicted in figure 1.2). It should be noted that to make this possible, it is necessary that two conditions are met: first, that values related to the development of a software project may be quantified and second, that we have the ability to interpret and classify the data. Once this has been solved, we should be in a position where we could identify those parameters that show *unhealthy* trends and that should be corrected. Of course, for the fulfillment of the second condition we require a knowledge base which allows us to determine what values can be considered as *healthy*. In addition, knowledge about how changing these values in the right direction is needed. All this knowledge can only be obtained by the continuous experience acquired from the analysis of a large quantity of (libre) software projects.

This thesis targets specifically the first of the two conditions, the one that is concerned with the quantification of parameters with which we will be able to characterize a software project. Therefore, we will make an exhaustive study of the data sources in order to know the kind of data we are able to obtain and of the analyses that we can perform on them. Afterwards, data will be analyzed from various perspectives, with special interest in those that will allow us to better know, on the one hand,

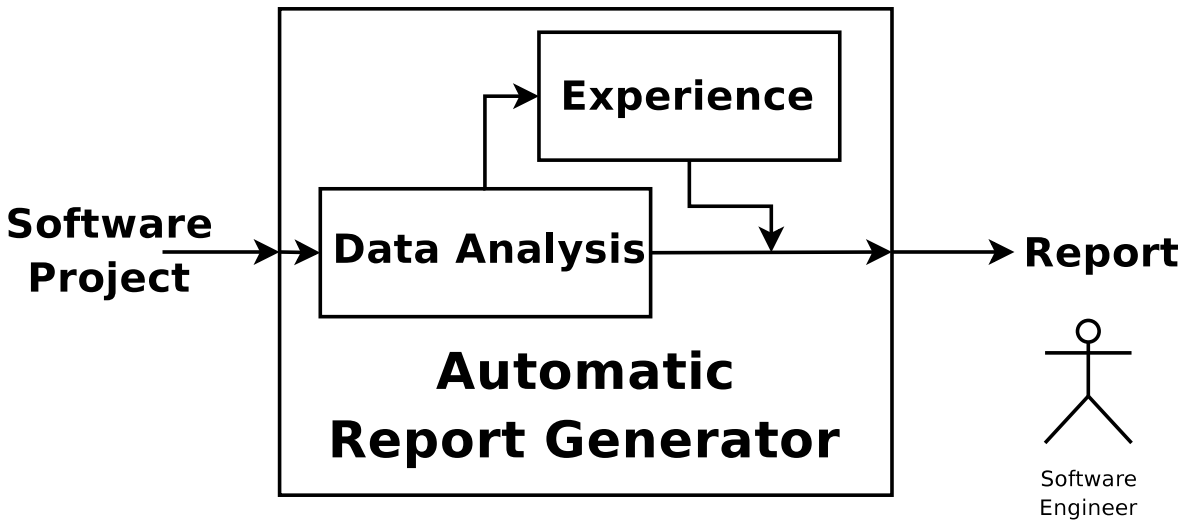


Figure 1.2: *The goal in the long run is to have such an automatic report generator. This thesis is heavily balanced towards the data analysis and all the tasks that it involves (description of data sources, extraction, methodologies, etc.), while the experience acquired should be increased in the future with the study of many more (libre) software projects.*

the innovative ways of organizing human resources in libre software, paying special attention to self-organization, distribution of tasks, creation of communities and social structure of a project. On the other hand, we are interested in technical aspects that may clarify the development process. We will therefore reproduce some of the *classical* software engineering studies. In addition, we will study some libre software projects and software compilations and how they evolve.

If possible, we will try that all the knowledge gained from our work -including the methodological one and the knowledge referred as lessons learned- could be shared by other members of the scientific community, and used in the own libre software projects. We look forward to being able to offer the data publicly (preserving the anonymity of the persons involved in the development) and to automatize as much as possible the extraction and analysis processes.

Although the main topic of this work is libre software, many of the methods that have been applied can be used in proprietary environments. This should be done in any case with care as some of the intrinsic characteristics of the way libre software is developed are not given in other domains.

1.4 Contributions of this thesis

The main contributions of this thesis can be summarized as follows:

- This thesis describes the data sources, the retrieval and extraction methods and the data itself that can be publicly found on the Internet about large libre software projects. The whole process has been summarized in a methodology and a set of tools, so that studies can be reproduced and enhanced by our and other research groups.

- This thesis is the first effort towards an exhaustive analysis of the libre software phenomenon. For this, a high number of libre software projects have been studied. But, above all, the methodologies and, especially, the tools used are the elements that make it possible to widen the analysis of all libre software projects using development and intercommunicating tools.

Other, more specific, contributions of this thesis are:

- It gives a detailed summary of the research literature on software engineering research concerning libre software and on the data sources that have been used so far (and how often they have been used) in mining software repositories (chapter 2).

- It verifies if large libre software applications follow the *laws* of software evolution (at least, one of them). Therefore, the growth of some large libre software systems over time will be studied and compared to *classical* assumptions (section 4.2).
- It studies the evolution of large libre software compilations taking into consideration the size of all software included, the number of packages, the number and share of programming languages, and other interesting parameters (section 4.3).
- It proposes a method to track and integrate information from developers from various sources that preserves their privacy and at the same time allows other research groups to benefit from the integration procedure and to publish the results in an anonymous way (section 3.7).
- It provides a new method of measuring the maintenance performed on a software project and discusses how we could use this information to infer the maintainability of that software in the future (sections 3.3.3 and 4.4).
- It studies the social structure of the communities of large libre software projects using techniques from social network analysis (section 4.6).
- It generalizes the concept of software evolution to other source artifacts that differ from source code files written in a programming language (for instance documentation, translation and user interface files) (sections 3.2.2 and 4.5).
- It analyzes the composition of the leading *core* group over time and how developers that belonging to it have been integrated into the project (section 4.7).
- It provides some lessons learned about libre software development, with special attention to large (in software size and number of contributors) libre software projects (sections 4.7 and 4.8).
- It presents a model that explains the libre software phenomenon from a macroscopic point of view using an analogy from biology (section 5.2).

1.5 Structure of the thesis

The structure of this thesis is as follows: chapter 2 is devoted to related research and a review of the papers that study the libre software phenomenon from a software engineering perspective. Other studies that are related to mining software repositories and to quantitative methods will also be introduced even if they do not target specifically libre software projects. Chapter 3 is a description of the data sources and the data that we can obtain from libre software project repositories. Although this part is already a main contribution of this thesis, it contains some ideas that have been already proposed by other authors; we could have included them in the chapter devoted to the related research, but have not done so for the sake of clarity and homogeneity.

Chapter 4 is the main body of this work. It contains a set of analyses and methodologies that can be applied to the data obtained and described in the previous one. The analyses range from technical-oriented ones to social aspects. Specifically, we perform a software evolution analysis on a large set of libre software projects, a software evolution analysis on a software compilation (a GNU/Linux distribution), a methodology that allows to see how much a software has been maintained and tries to infer its future maintainability, a file-type based analysis that attempts to identify the organization of tasks in the development of libre software, an analysis of the evolution of contributors (mainly volunteers) of libre software projects; and finally, a study that researches how members that hold currently a leading position are integrated into the project.

Chapter 5, includes the lessons learned from the study of the data sources and the set of analysis performed in chapter 4 to some large libre software projects used as case studies. It also includes a model that tries to explain on a macroscopic way the processes in libre software projects. This model will be verified against quantitative data obtained from results presented in literature and in this thesis. Finally, chapter 6 contains the conclusions of this work.

Chapter 2

Related Research

> *Research is to see what everybody else has seen, and to think what nobody else has thought.*
> Albert Szent-Gyoergi

This chapter is devoted to give a historical perspective of software engineering research on the libre software phenomenon. We will start presenting some first approaches to the development processes from outside the software engineering academic scope in the late nineties, then show some influences from *traditional* software engineering literature and finally give a detailed description of related research.

The main objective of software engineers that have studied the libre software phenomenon has been to get a *global picture* of the development and associated processes that take place. This picture is not limited to technical aspects; social issues such as organization, governance, decision structure, integration and task assignment are also of great interest. So, besides software engineers, researchers from many other areas have been interested in the last years in the libre software phenomenon - mainly sociologists, psychologists and economists.

There have been many efforts that study single projects and some research papers even have tried to do analyses on a large set of projects. But cases of detailed studies of large collections of projects (in the hundreds), crossing information from several different kinds of repositories, or analysis of historic patterns in different projects over long periods of time, are still rare.

This chapter has been divided in five sections: the first section gives historical evidence of such techniques even before using them on libre software; in other words, even though classical environments did not provide with such a rich set of data, there were already some efforts in that direction. The second one will be centered on technical details of the data retrieval process, presenting techniques and tools that can be found in the literature. The third section will be devoted to empirical software engineering studies. Some of the investigations that we will see there are classical studies that have been reproduced for libre software environments such as software maintenance, software evolution, software comprehension, the study of product families or field studies. Other investigations such as the study of software compilations or holistic (ecology) studies are technical-related, but have not been that frequent in classical literature. The fourth section is devoted to human-related and community issues. In this sense, libre software projects have been specially innovative, as flexible procedures and self-organization are the rule. In addition, most libre software developers are volunteers and do self-selected tasks. This provides with a large set of questions around developers that some research groups have tried to answer. Finally, the fifth section summarizes the data sources that have been used in literature.

2.1 The precursors: first papers about libre software development

Although libre software can be traced to the 1980s and even earlier [Salus, 2005], it became a matter of study only in the late 1990s. First voices were not from scholar researchers, but from participants in the libre software community itself who started to publish research results about how libre software

was developed, or about its economic consequences. Only around the year 2000, academic papers started to appear in workshops and congresses, and ended later in research journals and magazines. In the following sections we will briefly review the pre-academic era of research on libre software, which has had a high influence in later developments, when more sophisticated and methodologically organized research began.

2.1.1 The cathedral and the bazaar

The most prominent and influential text of this time is for sure *The Cathedral and the Bazaar* [Raymond, 1998], the well known essay by Eric S. Raymond. As one of the first attempts to characterize *the* libre software development model, it is based on the experience of the author. Although it is not supported by an empirical study of a meaningful quantity of projects, it reports several peculiarities present in many libre software projects, and drew the attention from many people who started to perceive something peculiar and worth studying in the libre software development. This work is at the same time an essay and a detailed case study of two libre software projects: the Linux kernel originally developed by Linus Torvalds and an own project named fetchmail, a small email-retrieval utility.

Before that essay, most of the literature about libre software had been concerned with the ethical and philosophical aspects ([Stallman, 1999], although published later, is a good example), or in the use of applications, with little attention to the development process itself. After an extensive research on the literature on libre software development, we have found almost no significant references before 1998.

The most important contribution of the *Cathedral and the Bazaar* is the creation of a metaphor. In it, classical software development models are modeled as cathedrals, in Raymond's own words "carefully crafted by individual wizards or small bands of mages working in splendid isolation, with no beat to be released before its time". Raymond does not only include here *heavy* development processes used in industry by software companies, but also some libre software projects from the GNU project, such as the GNU Compiler Collection (GCC) or the GNU Emacs editor. The development process is, in such cases, centralized and highly dependent on a few persons whose roles are tightly defined. Release management follows a closed planning, user feedback is limited and contributions by external developers are not fostered.

In opposition to the *cathedral* we find the *bazaar*, summarized as "Linus Torvald's style of development - release early and often, delegate everything you can, be open to the point of promiscuity". Raymond offers a list of best practices he has extracted from experience with the development of the Linux kernel. The most important lessons have been summarized in following list:

- Successful libre software projects start by *scratching a developer's itch* (lesson 1). This implicitly states two ideas: the first one is that there exists a lack of planning in the first stages. A developer has a need and enough knowledge to create a software to meet this need. No requirements have to be taken from other users, no misunderstandings will arise as the end-user is the developer himself. Second, it gives an idea of why the governance and the decisions in libre software are oriented towards those who actually code[1]. We will see in later sections (see 2.3.5) that this has changed since Raymond wrote his essay.

- "Plan to throw one away; you will anyhow." (Fred Brooks, MMM, Chapter 11 [Frederick P. Brooks, 1978]) (lesson 3). The prototypic way of developing in libre software is summarized here.

- The software should be released early and often (lesson 7). The rational of the *release early* practice obeys to the rule that a developer should publish a first, small but well-structured version to attract other developers who might have the same need. The purpose of the *release often rule* is to have a fast cyclical spiral development process. Mathematical models have shown that frequent versions foster feedback and minor bug-fixing times (in the mean) [Challet & Du, 2003].

[1] "Show me the code" is one of Linus Torvald's most known phrases when discussions appear in the mailing lists.

- Tightly tied to the previous point is that users should be treated as co-developers (lesson 6). Such an attitude may encourage participation in the project and imply rapid code improvement and effective debugging. This is summarized in *Linus' Law* which says that "given a large enough beta-tester and co-developer base, almost every problem will be characterized quickly and the fix obvious to someone" (lesson 8). The community (and its size) matters. *Linus' Law* could be reworded as debugging being a task that can be made efficiently in parallel, so that limitations given by Brooks' Law [Frederick P. Brooks, 1978] do not attend. The rationale for this is that less communication among developers while testing and debugging a software is needed.

- There should be a high emphasis in rewriting and reusing code (lesson 2). A good code base will enhance the quality of the software and make a lower barrier of entry possible. Other developers will be more inclined to join. The software license allows to reuse code from other projects, that usually has already been debugged and enhanced, so this task has not to be done by the development team.

- Finally, if the original author (and usually project leader) loses interest in the software, his last duty is to find a successor for the project (lesson 5).

The other main idea of this work is the importance of having communities around software projects. Raymond argues that while coding is an activity that remains essentially solitary, successful software comes "from harnessing the attention and brainpower of entire communities". Those who create projects which are *open* and enable the possibility of submitting feedback, code contributions or external debugging have a clear advantage over developers in *closed* source projects.

Besides its influence, the geographical dispersion and the intervention of many different actors in the development of libre software has made it a phenomenon difficult to study. Some statements have been taken for granted even if no data supports them. It seems from the reading of essays as *The Cathedral and The Bazaar* that a kind of *magic* is involved in the process. This is, of course, scientifically not acceptable. It is a duty of software engineering researchers to try to understand the development processes in detail and to provide ways to adapt the lessons learned to other projects.

Subsequent research has worked on Raymond's ideas and has tried to conceptualize in a more formal way the lessons learned depicted in his seminal work. In this sense, Senyard and Michlmayr define the set of activities for a libre software project to become a bazaar-driven project as the following [Senyard & Michlmayr, 2004]:

1. A prototype with plausible promise. Attention from the libre software community should be gained in the early stages of the project by a small but functional software.

2. Modular design. Such a characteristic will allow to lower the communication requirements among the development team. Developers or developer teams may work independently on a part of the project without having many interferences from the other groups.

3. Source code available, usable (compiles and executes) and workable. As a modular design is pretended, the software will be composed of many parts. To avoid the problems that arise when integrating all components at the end of the development phase, there should be always a working version that is downloadable and usable without much effort. This lowers the barrier of entry for new users and co-developers.

4. Community surrounding the project. Users and co-developers are the ones who will make the project evolve. A small community will not be very attractive to new users and co-developers. Building a community is based on a network effect: attracting users and co-developers will attract more users and co-developers. Entering such a network effect is one of the most relevant goals of companies involved in the development of libre software and thus raises many research questions.

5. Communication and contribution mechanisms, both technical and human. Although communication among the project members is tried to be minimized by modularization, it cannot

be avoided. In fact, the communication and contribution mechanisms are one of the main parts of libre software development and have to be managed both technically (by tools that have become *standard* in the development of libre software as mailing lists, bug-tracking system or versioning systems) and from a social perspective (information managers, web administrator, among others).

6. Well-defined scope of the project. The *UNIX culture* of having small and specific-purpose programs which may be used together with other tools is still valid.

7. Defined coding standards or style. The objective is to help code reading to developers that form the developer group and to ease the contributions of external developers.

8. Attractive license for external contributions. Some licenses do not allow to integrate code from other libre software projects easily or may have tricky clauses.

9. Suitable management style. The license of the project allows to create a new, independent project based on the project's source code. This is, of course, an uninteresting situation as the intention is to embrace as many external contributions as possible, and having several projects with the same source code base may *split* the community. Hence, a *balance* should exist between taking (too much) control over the project and providing enough freedom to contributors. A wide range of scenarios can be observed in the libre software world, that range from the rigid management style of the Linux kernel [Iannacci, 2003] to other more loose styles as those in use at KDE or GNOME [Germán, 2004b].

10. Appropriate amount of project documentation. Software, code and processes should be properly documented to facilitate the use of the software and collaboration of new developers.

This list has been given from the historical experience in many large libre software systems, but it is by no way a recipe for achieving a successful libre software project. The authors note that the first six are crucial, while the seventh to eleventh are important and the last one is desirable.

The interest that libre software has gained in recent times among industry has also had as an effect that some already existing proprietary software (sometimes up to several million lines of code) has been released under a libre software license, not fulfilling the first condition. This is the case of the Mozilla Internet suite released by Netscape or the OpenOffice.org office suite by Sun Microsystems.

2.1.2 Early follow-ups

The influence of *The Cathedral and the Bazaar* has been large. Soon, other authors have built on top of it, or have criticized its main results. Probably the first well known follow-up came from Bezroukov [Bezroukov, 1997] whose main point was that the bazaar development model is not a revolutionary phenomenon, but just another form of "scientific community". Bezroukov considers Linux (as a development model) to be just the natural evolution of the practices found in the GNU project. All in all, Raymond's ideas are deemed as too much simple to match reality, and the article proposes some other models based on academic research that would explain the phenomenon better.

Another perspective is provided by Vixie, who compares the classical waterfall model with the processes that are used in the production of libre software [Vixie, 1999]. While the first one is composed of a set of complete and logical sequence of steps with the goal of obtaining a final product, the second relies on a lack of formal methods, mainly avoided because they are not satisfactory for programmers who voluntarily devote their time to programming. In the opinion of the author, lack of formality is compensated by user feedback, and by the introduction of software engineering methods when developers face some specific problems for which they are experienced enough to define a methodology. Therefore, no *ad-hoc* development model for libre software projects exists; the methods and practices evolve with the project and usually show a tendency to the formalization of tasks.

In addition to these studies, many others were published during the early 2000s. A good, in depth description of the state of the art in research about libre software by that time is the book by Feller and Fitzgerald [Feller & Fitzgerald, 2002], which provides a good view of the landscape of early models and learnings after Raymond's essay.

2.1.3 Towards empirical-based studies

All studies presented so far have tried to obtain qualitative conclusions from impressions gained from a limited number of libre software projects. Soon, the limitations of these approaches were identified and a more traditional, approach, based on quantitative data was adopted by many research groups.

Although we will see that before 2002 there were also some empirical studies of this kind, it is that year that can be considered as the starting point of a common trend. It was at that point over time when several research groups realized that the complexity of the libre software phenomenon cannot be understood without previously acquiring factual data from projects. The main reasons for this trend were the great diversity of the libre software development practices observed from project to project (which made generalizations risky), and the promising research path of first empirical studies.

There were also influences from other communities that were not interested in libre software *per se*, but saw in the availability of data an opportunity of obtaining evidence that could validate some preexisting theories and models. In this area, it is worth mentioning the community interested in mining software repositories[2], with researchers coming from various software engineering fields: machine learning, software evolution, software effort calculation, software maintenance, software reengineering and other.

2.2 Mining data sources

An important part of this thesis is devoted to the identification and description of the data sources that libre software projects offer publicly. Although being a relatively new field of research, much related work exists. The mining software literature has converted itself in some cases in an end in itself, and not the means to an end. Thus, some of the research activities have been more focused on how to mine the available data sources with information about libre software projects than in analyzing such data to get higher level information. In other cases, the lessons learned have been of relative little importance compared with the understanding of the mining process.

The next subsection describes the main lines of the methodology usually applied, with different degrees of fidelity. After it, we will present the most used data sources in literature, with special attention to the methods that have been applied to extract, clean and normalize the data set. In addition, methods of fact extraction and the most common used metrics will be presented.

2.2.1 Methodology

Gasser et al. postulate the set of characteristics that empirical approaches should consider when mining software repositories [Gasser *et al.*, 2004]: (1) direct reflection of reality, (2) adequate coverage, (3) examination of representative levels of variance, (4) demonstration of adequate statistical significance, (5) comparability across projects, (6) repeatability, and (7) testability and evaluability of the provided results.

In addition, four not independent main elements can be identified for these studies: software artifacts, software processes, development communities and knowledge of the participants.

Therefore, any infrastructure that allows for this kind of empirical research should include data sets about libre software development elements that

- are of an empirical and natural origin (this assumption targets characteristics 1 and 2),
- are of a size large enough (characteristics 2, 3 and 4), and
- offer the possibility of sharing data in common frameworks and representation (characteristics 5 and 6).

The nature of the empirical data may vary substantially. It may be content (which includes communications, documentation and development data), media sources (i.e. communications systems

[2] A good sample of this community can be found in the Mining Software Repositories workshop, co-located with ICSE, see http://msr.uwaterloo.ca/msr2004/ and http://msr.uwaterloo.ca/msr2005/.

or versioning systems, among others) or locations (as community websites, software repositories, and other). Interestingly enough these data are the byproduct of the development, maintenance and other project-related activities.

[Gasser *et al.*, 2004]] also point out that data identification and preparation are key elements of the research process, a process that has some associated difficulties. This process is in fact the implementation of a methodology, which has following phases:

1. The discovery and proper selection of the data.
2. The access and gathering of the identified data.
3. The cleaning and normalization of the obtained data.
4. The enrichment of the data by aggregating links from another data sets, or by relating data sets at different points over time.

In addition, standards for the representation of factual information should be adopted, to avoid the dependency of the different services from which the data sets are obtained (for instance, the many different versioning systems, or bug report systems, used in libre software projects).

2.2.2 Data sources

Any quantitative analysis has to start by identifying the data sources to be used. In the case of libre software, fortunately many public repositories of information about the development process exist. All these repositories are potential data sources. In a first approximation, the following rough classification of those data sources can be considered:

- Data obtained from the products that are the result of the development process. Mainly this affects source code, and sometimes the binary packages. Sometimes it includes documentation, graphics and multimedia materials, GUI [3] design files, among others.
- Data obtained from tools used during the development process. Most projects use a versioning system (usually a version control system, such as CVS or Subversion). This allows to retrieve the current and past state of the source code and to obtain additional information on the changes to the software: who has done them, when they have been done and other meta-data.
- Data obtained from the systems used to support communication among actors in the development process. This includes mainly the information in mailing lists and forums used by developers to communicate among themselves and with other parties interested in the project. But also some other systems such as the usual bug control system, used to keep track of the communication of bug reporters and developers, can be considered.

In most cases, all these data can be obtained from the same Internet site: the hosting site for the project. Even hosting sites that provide facilities for thousands, or tens of thousands of projects exist. This allows for the easy and automatized retrieval of data for huge quantities of projects. Some of the most well known sites of this kind are SourceForge[4], BerliOS[5], Savannah[6] and Alioth[7], although many other have appeared in the last years. Some other projects, on the contrary, prefer to maintain their own support site (especially when they are large enough in size, or when the project is driven by a company). Nevertheless, they usually provide facilities of the same kind, and are also automatically searchable and downloadable.

The research community has found in them a particularly rich source of data, as traces from using the various tools can be linked together in many cases. But some intrinsic problems when mining these repositories arise. Among others, the following can be highlighted [Howison & Crowston, 2004]:

[3] GUI is the acronym for graphical user interface.

[4] SourceForge is the largest and most popular development-supporting site: http://www.sourceforge.net

[5] BerliOS is a German-based site based on the SourceForge software: http://www.berlios.de

[6] Savannah is a project lead by the Free Software Foundation (also based on an enhanced version of the SourceForge software): http://savannah.gnu.org

[7] Alioth is the development platform offered by the Debian project: http://alioth.debian.org

- The first set of problems is related to how to spider these sites. In most cases, access is not granted to the database which stores the information, and all that can be done is to download the HTML pages offered to the public. This means that tools that retrieve all the information from the web have to take special care in not overloading the servers (for instance by waiting after every single download). Even so, the risk of being banned exists.

- After spidering and storing the pages locally (which is highly recommended), parsing starts. Here problems with unexpected characters, such as non-ASCII characters or HTML entities, are specially perilous. On the other hand, trivial changes in the layout may make the parsing coded not useful anymore.

- Incremental parsing is often not possible. This means that getting an update on the state of a project, all the web pages with the data about that project have to be download again.

Sometimes XML or *backend-type* interfaces simplify this spidering activity. Using some of this information and other techniques, the OSSMole[8] project has become a good example of what kind of information can be obtained from a site like SourceForge (more about this will be introduced in subsection 2.2.3).

The following subsections provide a summary about what information is being obtained from the different data sources.

Source code

Source code is the main output of a software project, and has been a matter of study since the beginnings of software development. Quantitative analyses of source code have for instance being of paramount importance for effort estimation and complexity studies. Some of the data that can be obtained from source code are:

- Lines of code, which come in different flavors. We can just count all the lines in files with source code, or can use for more specialized concepts, such as physical or logical SLOC (source lines of code[9]). SLOCs are more difficult to calculate since they are language-dependent, but they are also more useful in many contexts.

- Complexity metrics, such as McCabe [McCabe, 1976], Halstead [Halstead, 1977] or more recent object-oriented metrics [Zuse, 1991].

- Authorship information, which can usually be extracted from copyright notes in the source code. In the case of libre software, those notes allow for the identification of individuals related to the development of the code, and often to companies involved.

- Structural relationships. In most programming languages source code has indications that allow to track dependencies and relationships among modules in a software package.

- Categories of source code. Sources of software systems do not contain only code written in a programming language. Usually, some other artifacts, such as documentation files, translation files or user interface definitions, can be identified and analyzed.

This kind of information can be obtained at different granularity levels, and be analyzed together for a given software package or even a software distribution which integrates huge numbers of packages (at a *macroscopic* level), or be scrutinized at the *microscopic* level. In addition, source code is usually stored in a versioning system, which allows to retrieve snapshots of the software at any point in the past. This is, of course, a good starting point for historical analysis on the evolution of any of the former kind of information.

[8] http://ossmole.sourceforge.net

[9] SLOC is defined as "a line that finishes in a mark of new line or a mark of end of file, and that contains at least a character that is not a blank space nor comment".

Versioning systems

Versioning systems store records about any change performed, with detailed information of who modified what parts of the code, and when. Fortunately, most libre software projects, and specially the large ones, maintain public versioning system repositories, from which this information can be easily extracted. This way the actions of developers can be tracked and related to the changes in the code.

The most used versioning systems is CVS which can be seen as the *de facto* standard in the libre software world (although some new generation tools, such as Subversion or Arch are substituting it slowly). The usual methodology followed to get development information from CVS is described, for instance, in [Zimmermann & Weissgerber, 2004] (for other tools the process can be somewhat different):

- The first step deals with data extraction. Once we have obtained the raw information of the module we are analyzing (usually by using the *cvs log* command), it is stored in a database. This way information about all files and directories, revisions, tags[10] and branches[11] can be obtained.
- The second step is aimed at restoring transactions. CVS does not feature the concept of *atomic modification*[12] (when changes on several files are done in the same transaction). Several algorithms to identify those transactions exist, such as those based on fixed or sliding time windows [Zimmermann & Weissgerber, 2004; Germán, 2004a; 2004b].
- The third step deals with further fine-detail analysis; the file-level granularity that CVS logs provide is often not enough when the researcher is interested in changes at the module or function level.
- A final step is devoted to data cleaning, including the identification of large transactions that usually do not provide further insight to the object of study (for instance, changing the year in the copyright statement in the top of files implies usually having a large transaction including all files in the source code base).

Despite the high quantity of relevant data that can be obtained from CVS archives and similar repositories, and the many studies published in this area, still some practical open challenges can be pointed out. In [Germán, 2004c] some of them are enounced: lack of common terminology in the published studies, difficulty of using the CVS servers of some projects (for instance, because of the stress put on them), and the need for test cases for tools being developed in this domain which would help to homogenize and validate results.

Bug Tracking Systems

Bug tracking systems (BTS) are used to collect bug reports and feature enhancement petitions from users and developers through a web interface. The most known BTS is the one developed by the Mozilla project and called BugZilla[13]. A bug-tracking system has several goals: first, it is an attempt to ease the user feedback, so that it is complete and usable by the developer (it should be noted that users do not have to know what type of information is important to solve their problem as they do not know the internals of the software). Second, it allows to properly organize all requests by the development team; if the project is too large, then this task may be taken over by a professional as it is done for large projects as GNOME or Mozilla[14]. And last, it permits to track the number of errors and their importance in the project and for any application or component that is part of it.

[10] A tag is a text label associated with a version of a file. It is generally used to specify all files that belong to a release.

[11] Branches are lines of development that diverge from the primary line of development. Branches allow for parallel development; so a stable branch, where only errors are corrected, may coexist with a development one, where functionality may be added.

[12] Atomic modifications can also be considered to be as *modification requests*, a concept that is found in some papers in research literature [Mockus & Votta, 2000; Eick *et al.*, 2001; Mockus *et al.*, 2002; Herbsleb *et al.*, 2001].

[13] http://www.bugzilla.org

[14] A brief description of the tasks and lessons learned by the *bugmaster* of the GNOME project can be found in [Villa, 2003] and [Villa, 2005].

Each bug is given a unique number, so that it can be easily identified by developers. Some research papers and tools use this unique number to link bugs in the bug-tracking system with the versioning system [Mockus *et al.*, 2002; Germán & Mockus, 2003], as we will see in section 3.3.1.

Usually the database behind any BTS is not available for public querying, which means that without special arrangings with the projects, the researcher can only access to the web interface (usually through HTML pages). Unfortunately this means a much higher stress on the BTS server, and usually the impossibility of downloading information incrementally (that is, getting all the changes to bug reports after a certain date).

Communication exchange

Although direct (face to face) communication among the members of a libre software can happen, in most cases it is not usual. On the contrary, the preferred way of exchanging information are multi-cast systems such as mailing lists or (web-based) forums in an asynchronous mode and Internet Relay Chat (IRC) or Instant Messaging (IM) services for synchronous communications. Synchronous communications could be stored and made publicly available, but this is not the usual situation. For mailing lists and forums archival is done and, if available publicly, artifacts can be retrieved and analyzed.

Mailing lists suppose the main decision and information exchange vehicle in libre software projects, although if a project gets larger in size usually other complementary means are introduced [Germán, 2004b].

Meta-data and other sources

Some other data sources have been mined. These are usually information sources that are not *traditional* sources and that have been used in a minority of research studies. In any case, we include them to give a broad perspective of the possibilities that exist beyond the ones presented above.

One of them are the changelogs that are used by developers for reporting in inverse chronological order the changes that have been performed on the sources. Although changelogs do not have meta-data and tag-syntax associated which would make them easy to parse, a GNU standard that is more or less popular among libre software projects exists [Chen *et al.*, 2004]. We can find different approaches of data extracted from changelogs by Capiluppi [Capiluppi *et al.*, 2003], Tuomi [Tuomi, 2004], Germán [Germán & Mockus, 2003] and Chen et al. [Chen *et al.*, 2004].

Another data source that has been used are the Linux Software Map entries. The Linux Software Map is a database of software written or ported to Linux[15]. Applications and tools in that database are described in a structured way, allowing to extract information easily about them as for instance the maintainer name, the primary site of the software, its license, among others. Dempsey et al. [Dempsey *et al.*, 1999] used this information to analyze the geographical distribution of developers.

Finally, some authors opine that text should be treated as software and that the analysis software repositories should be routinely augmented with the language text which is used to develop the software systems [Dekhtyar *et al.*, 2004].

2.2.3 Tools

Many tools can be used to retrieve information from the described data sources. Many of them are themselves libre software, and therefore not only freely available to the research community, but can also be enhanced or adapted to the specific needs of any research team. In this subsection, some of them are described.

SLOCCount

SLOCCount[16] is a tool authored by David Wheeler that gives the number of physical source lines of code of a software program. SLOCCount takes as input a directory where the sources are stored,

[15] More information about the Linux Software Map can be found at http://lsm.execpc.com/lsm/.

[16] http://www.dwheeler.com/sloccount/

identifies (by a series of heuristics) the files that contain source code, recognizes for each of them (also by means of heuristics) the programming language, and finally counts the number of source lines of code they contain. SLOCS are parsed differently for different languages, which forces the identification of programming languages.

SLOCCount also identifies identical files (by using MD5 hashes), and includes heuristics to detect (and avoid counting) automatically generated code. These mechanisms are helpful when analyzing the code, but have some deficiencies. Finding almost identical files using such hashes is not very effective. In the second case, heuristics only take care of well-known and/or common cases, but do not detect all of them, or others that may appear in future. Finally, SLOCCount uses its counting for calculating the cost estimation that the Basic COCOMO model produces [Boehm, 1981].

SLOCCount has been used on studies on Red Hat [Wheeler, 2001] and on Debian [González-Barahona *et al.*, 2001] as we will see in section 2.3.3.

SoftChange

SoftChange[17] is a tool, developed using a *clean room* method by Daniel Germán and Audris Mockus, that summarizes and analyzes software changes in versioning repositories (specifically CVS) and bug tracking systems (specifically BugZilla and GNATS). The architecture of this tool can be split into three components: the trails extractor which extracts software modification trails and inserts them into a database, the fact enhancer that generates new facts from the data that had been stored in the database and finally the visualizer which provides results through a web interface.

SoftChange uses a sliding window algorithm to rebuild atomic commits (the authors use originally the term *modification requests*) and also performs an analysis of the changelogs, mailing lists and the BugZilla bug tracking system. One of its features is the linkage of bugs from BugZilla and the atomic commits in CVS by looking for bug ids in the CVS logs using regular expressions. This tool is described in depth in [Germán, 2004c; Germán & Hindle, 2005].

OSSMole

OSSMole[18] is a project that pretends to provide raw data about libre software projects, mainly from web-based development sites as SourceForge. The aim of the project goes farther beyond of being a tool that extracts and analyzes a data set, as it provides the data it extracts and looks for *donated* data sets from other research groups.

Basically, OSSMole retrieves the web pages with development information from the web-driven development platform and and parses the web pages. The data obtained are stored into the database. As the amount of projects that are hosted in SourceForge (the most popular development platform) is in the order of the 100,000, and web page request flooding has to be avoided, the retrieval process may take weeks [Conklin *et al.*, 2005]. Using machines in parallel helps to lower this time span. OSSMole is thus a tool that allows to analyze large amounts of libre software projects.

PieSpy

PieSpy[19] is a tool that monitors IRC channels to find social relations among participants in those channels. It is based on a set of heuristics that identify personal, but non-private conversations in a channel. Once the links have been obtained, social network analysis can be done including the graphical display of the relationships or animations with the evolving social network [Mutton, 2003].

Other tools

The following tools, although not specialized in *milking* project repositories, can also be used for some kind of static analysis on the source code of any software project:

[17] http://sourcechange.sourceforge.net/
[18] http://osssmole.sourceforge.net
[19] http://www.jibble.org/piespy/

- CCCC[20] stands from C and C++ Code Counter and is a tool that analyzes C++ and Java files and generates a report on various metrics of the code. Metrics supported include lines of code, McCabe's complexity [McCabe, 1976] and metrics proposed by Chidamber & Kemerer [Chidamber & Kemerer, 1994; Hitz & Montazeri, 1996] and Henry & Kafura [Henry *et al.*, 1981].

- LOCC[21] supports hierarchical and incremental measurements that helps in estimation, planning and other software engineering activities. The *hierarchical* size measurements refer to the number of packages, classes, methods and lines of non-comment source code per method.

In addition, Chris Lott[22] lists various static code analysis tools that compute metrics defined on C and C++ source code. The metrics are primarily size and complexity of various types (lines of code, Halstead [Halstead, 1977], McCabe [McCabe, 1976], among others).

2.2.4 Exchange formats and repositories

Tools are just a first step for making the research process as automatic as possible. As we have seen in the previous subsection, we have in this regard already some solutions for many of the data sources. But tools are only a means for acquiring the data that will be the basis of subsequent analysis and research activities. So, having the possibility of accessing data once it has been retrieved, extracted, cleaned, normalized and possibly enriched will provide researchers with further possibilities.

This is where exchange formats and repositories with raw or *cooked* data about libre software development projects come in. Current state of the art in the field of libre software engineering and even in the software engineering field is not very advanced to the knowledge of the author. So, first attempts in this direction have used ideas from other areas.

In this line, probably the most used format today for the exchange of data (maybe after CSV or SQL formats, which are not specifically oriented to this use) is ARFF[23] (Attribute-Relation File Format). ARFF files are ASCII text files that describe a list of instances sharing a set of attributes. The format has been designed by the Machine Learning Project at the Department of Computer Science of the University of Waikato for use with the Weka[24] machine learning software. One of the most well known archives of information about software development (in general, not specific to libre software) using extensively this format is the PROMISE repository[25] [Sayyad Shirabad & Menzies, 2005].

2.2.5 Integration of data from different sources

Libre software provides the possibility to study many traces left behind during the development process. But unfortunately, the information is not structured in such a way that makes it easy to interconnect traces related to the same artifact but coming from different sources. Therefore, methods for the integration of the information gathered from several places have to be found. In this respect, several possibilities have been explored, which could also be used in union:

- Integration through artifacts. We define a granularity level (project, directory, file, class, method or even line) at which we identify all the actions related to every artifact.

- Integration by identification of traces from other sources. For instance we may find bug report identifiers in version repository logs. Some other research groups already have started to walk in this direction [Antoniol *et al.*, 2005; Fischer *et al.*, 2003; Mockus *et al.*, 2002; Germán & Mockus, 2003].

[20] http://cccc.sourceforge.net/
[21] http://csdl.ics.hawaii.edu/Tools/LOCC/LOCC.html
[22] http://www.chris-lott.org/resources/cmetrics/
[23] For a detailed description of the ARFF format, visit http://www.cs.waikato.ac.nz/ ml/weka/arff.html
[24] Weka is a libre software that allows to perform a collection of machine learning algorithms for data mining tasks. More information can be found at: http://www.cs.waikato.ac.nz/ml/weka/.
[25] http://promise.site.uottawa.ca/SERepository/datasets-page.html

- Integration at the developer level. Actors that take part in the development process can be identified, and their activity tracked in the various sources of information, even when different identities are used (several e-mail address, logins, and even different spelling of real names).

In this thesis we will propose a methodology for integrating data at the developer level preserving at the same time their privacy (see section 3.7).

2.3 Empirical software engineering studies

This section contains a summary of the technical analyses that have been done on libre software projects. We will see that literature includes various angles of the libre software phenomenon. Studies range from maintainability, evolution, product families and compilations, holistic and fine-grained analyses to some field studies. Most of them have been performed and published in recent years and many of them are simultaneous work to this thesis.

2.3.1 Historical influences

The use of historical data from software development is not a new practice in the software engineering community. But most of the approaches that have used such data have focused on evolutionary aspects [Lehman & Belady, 1985; Gall *et al.*, 1997; Mockus & Votta, 2000] or on looking at the reasons for the most common behaviors found in a software development project [Mockus & Votta, 2000; Eick *et al.*, 2001]. Of course, availability of the source code as well as the public access to software tools used during the development process in libre software projects have made them gain attention in recent time.

Already in the late 90s, some research groups had access to the versioning repositories of large industrial systems. In this sense, Gall et al. demonstrated from their study of a multi-million telecommunication software that the evolution of the system over time as a whole may mask the evolution of its subsystems that may have completely different behaviors [Gall *et al.*, 1997]. The authors maintain that having a database with the product releases would be very valuable for the management of software projects, specially regarding planning future schedules or estimating maintenance costs.

Some others have looked at the versioning system, attending to aspects that are beyond the source code, such as the influence of software tools on the development process [Atkins *et al.*, 1999; 2002] or the identification of the reasons for changing the software [Mockus & Votta, 2000].

The *human* component may be used to infer the development and maintenance costs of the software development life cycle. Some works have already studied this aspect, trying to figure out how much effort is to be applied when performing changes to a software by studying software repositories [Graves & Mockus, 1998]. In this regard, Herbsleb et al. have studied the implications of having distributed teams working together in a software project [Herbsleb *et al.*, 2001]. Although this study is based on a survey responded by employees at a traditional software company located in several locations around the globe, it also makes use of historic development data in form of modification request (a formalized way that is used to requests the incorporation of specific functionality into the software).

The main limitation of the literature presented so far is that in general it has only studied a small set of case studies, in most cases only one, even if some of them have been very large in size. This raises the question of their significance and their applicability to other software projects (especially to libre software projects, as their development style is in general completely different). In addition, the methodologies that have been used in these studies were mainly *ad-hoc* and very specific as they make use of project-related technical and management details that makes their reproduction in other projects difficult.

2.3.2 Software maintenance and evolution

Software maintenance is the last phase of the software life-cycle. It is usually also the longest over time and the one that is more costly; Conger suggests a 20-80 rule which supposes that 20% of the

effort is devoted to development and 80% to maintenance activities [Conger, 1994]. The availability of source code in libre software projects has open the possibility of new studies with a large amount of projects in many languages and for different environments and final uses. This is an advantage not only from the quantitative and empirical point of view, but allows for new ways of learning from already existing code through code reading [Spinellis, 2003].

One of the first studies of software maintenance of libre software projects examined the Linux kernel for its maintainability [Schach *et al.*, 2002]. The authors define maintainability as the inverse of common (also known as global) coupling between the main kernel modules and all other modules. The findings show that the number of couplings grows exponentially with version number, while growth in lines of code for kernel modules is only linear. In the authors words a restructuring is needed as the maintenance of Linux will suffer in the next future if such a trend is maintained.

Another study on code maintainability in libre software projects studies almost six million lines of code [Samoladas *et al.*, 2004]. The authors have chosen for this study to use the Maintainability Index (MI), a composite metric proposed by Oman [Oman & Hagemeister, 1992; 1994]. They have found that generally libre software code has more quality than proprietary code. They suppose developer motivation is the key for such a pattern. But, as in the proprietary case, maintainability problems gain importance with age and preventive maintenance should be fostered.

Software maintenance from the study of software trails

Another approach to the reconstruction of the evolution of software is [Germán, 2004e], which uses software trails for that matter. Therefore the de-facto standard tools used in libre software development (CVS, mailing lists and changelogs) are analyzed and correlated. The author argues that although software trails do not tell the whole story, they give important information about the evolution of the software schema and of the informal structure of the project. The vast amount of information that is available makes the use of proper visualization tools and metrics a necessity. The results of the technique applied to the Evolution project[26] show the evolution in number of contributors and the file types that have supposed most work. Another outcome is that most files are rarely changed. Then the analysis is done on the components of the software application, throwing out that developers have a tendency to focus on specific parts of the software architecture (see figure 2.1 for the authorship graph). The author labels this as "developer territoriality".

The continuation of that study is a fine-grained analysis of the software trails in the CVS versioning system of a libre software project [Germán, 2004a]. Instead of using commits as the measure of activity, the author uses atomic commits. This allows him to categorize afterwards the atomic commits, selecting the ones that mostly contain source code files for further analysis. Atomic commits are classified according to their purpose: maintenance (defect fixing), improvement of functionality, documentation, evolution of the architecture (major changes in the API, etc.), code relocation or merging of branches. For the ones that mostly contain code, attention is put on those who correspond to bugs reported in the bug-tracking system and those where only comments in the source code files were touched.

The findings of this work are that most atomic commits contain very few files, being the existence of the changelog files an important skewing factor. Separate development and maintenance periods could be identified; maintenance periods have less atomic commits than those periods devoted to adding functionality. The number of changes done per atomic commit is also less in the former.

Rysselberghe et al. have proposed a method that helps in the study of the changes done on a software system hosted in a versioning system [Rysselberghe & Demeyer, 2004]. The authors display a horizontal line for every file and commits are given by dots in that line (the size of the dot is proportional to the number of lines in the commit). This allows to visually identify what parts of the system are changed over time and also what files do change simultaneously. As a drawback, the authors say that the way the lines are ordered should be enhanced; their proposal of doing it on a module basis and then in alphabetical order is just a first approximation.

[26] Appendix A contains a brief description of Evolution and of other libre software applications studied in this thesis and in related literature.

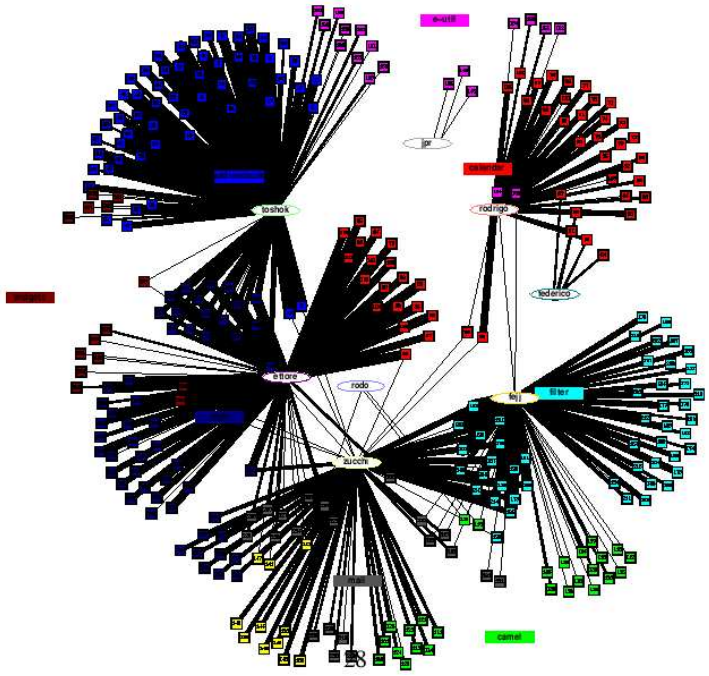


Figure 2.1: *Authorship graph giving the relationship among files (represented as squares) and the developers (given by ellipses) that work on them (*developer territoriality*). As it can be observed the clusters correspond to software components (rectangles). Source: [German, 2004a]*

Gîrba et al. base their work on the just introduced methodology [Gîrba *et al.*, 2005]. They aggregate some information that allows to identify what developers have developed what part of the system. They do this by introducing colors for authors and by assigning a color to horizontal lines depending on the author who has contributed most to a file. Hence, if red is assigned as the color of an author, all commits done by him will appear in that color. As well, all horizontal lines that represent files the commiter has contributed most will have that color. The authors solve the problem of displaying files properly by using the Hausdorff metric, so that the lines that correspond to files that are changed near in time also appear next. The results provide some patterns that appear during the development: monologue (a period where changes on a file are done by one author), dialog (changes are introduced by multiple authors), teamwork (a special case of dialog with a fast succession of changes by multiple authors), silence (a period with nearly no changes at all), takeover (a developer takes over a large amount of files initially *owned* by somebody else in a short amount of time), familiarization (accommodation over a long period of time - as opposed to the fast transition that takeover supposes), expansion (new files are added), cleaning (removal of some files), bug-fix (small localized changes) and edit (under which other non-functional changes fall, such as cleaning comments, changing the license, among others).

Zimmermann et al. have developed a methodology that helps in the maintenance of software systems by mining for previous software changes from the code base [Zimmermann *et al.*, 2005]. This approach is based on the idea that if we consider functions that have been changed at the same time, future changes to any of these functions are likely to have an influence on the others. This can be used hence to predict and suggest further changes, to show coupling and to prevent errors because changes are incomplete. This methodology is available as a tool that can be embedded into the Eclipse Java IDE.

Software archaeology and code decay

Parnas introduced the concept of software *aging*, creating an analogy between programs and persons; both get old with time [Parnas, 1994]. This means that programs lose their appeal and that maintenance becomes a burden, the same for good and bad programs. Although this process is unavoidable, it can be fought. Parnas gives some hints: it should be designed for change, to document

the code, to introduce third-party reviews, among other activities. Libre software is, of course, not different from *traditional* software in this regard and programs have to cope with the addition of new features, while removing bugs and errors that exist in the code.

In this thesis we will see a similar concept to Parnas' *aging*: software archaeology[27]. Although the concept of software archaeology is not new nor unknown to the software engineering community [Hunt & Thomas, 2002], there have been few studies regarding this issue from an empirical point of view.

The idea of using the concept of archaeology to software maintenance can be tracked at least to the OOPSLA 2001 Workshop on Software Archeology that was organized by Ward Cunningham et al. The promoters of this workshop had a set of assumptions, being the first one the rationale for using the archaeology concept:

> "[Software] [a]rch[a]eology is a useful metaphor: programmers try to understand what was in the minds of other developers using only the artifacts left behind. They are hampered because the artifacts were not created to communicate to the future, because only part of what was originally created has been preserved, and because relics from different eras are intermingled."[28]

The concept of software archaeology has been generally used for large old (legacy) systems, but it is valid for any type of software with independence of its age and size as we will see in section 4.4. The main point is that while maintaining a software, developers view code that may have been developed and changed in possibly many points over time and by many different developers.

A similar approach has been undertaken by Eick et al. in a paper studying *code decay* [Eick *et al.*, 2001]. The authors state that while software does not age or *wear out* in the conventional sense, system become more unmaintainable over time because of changes on the software. So, the conceptual model that is proposed is based on the assumption that "a unit of code is decayed if it is harder to change than it should be measured in terms of effort, interval and quality". The paper gives four main results which show a tendency to code decay: the span of changes increases over time, the modularity declines, introducing changes produces new faults which have to be corrected in subsequent changes and that time span and size of changes are important predictors at the feature level. They demonstrate that code decay is a generic phenomenon that affects all software systems which are complex, but that decay is mostly due to changes to the software and not to software complexity.

Software evolution

Thirty years of research on software evolution have resulted in a set of *laws*, known as Lehman's Laws of Software Evolution [Lehman & Belady, 1985; Lehman & Ramil, 2001]. Although the number of laws has grown from three in the seventies to eight in their latest version [Lehman *et al.*, 1997], all of them have been empirically proved by studying projects developed in traditional industrial software development environments.

Data obtained by Lehman and other fellow researchers from the late sixties up to late nineties from several large industrial systems, have been summarized in a set of *laws* of evolution, which in their latest form have been announced as follows [Lehman & Belady, 1985; Lehman *et al.*, 1997]:

1. Continuing Change - A system must be continually adapted else it becomes progressively less satisfactory in use.

2. Increasing Complexity - As a system is evolved its complexity increases unless work is done to maintain or reduce it.

3. Self Regulation - Global system evolution processes are self regulating.

4. Conservation of Organizational Stability - Unless feedback mechanisms are appropriately adjusted, average effective global activity rate in an evolving system tends to remain constant over product lifetime.

[27] In American English *archeology*. Comes from the Greek meaning αρχαιος (ancient) and λόγος (word/speech).

[28] This quote can be found at the OOPSLA 2001 Workshop of Software Archaeology web page: http://www.visibleworkings.com/archeology/.

5. Conservation of Familiarity - In general, the incremental growth and long term growth of systems tend to decline.

6. Continuing Growth - The functional capability of systems must be continually increased to maintain user satisfaction over the system lifetime.

7. Declining Quality - Unless rigorously adapted to take into account for changes in the operational environment, the quality of a system will appear to be declining.

8. Feedback System - Evolution processes are multi-level, multi-loop, multi-agent feedback systems.

These *laws* have been validated empirically with some large industrial software projects, but the number of empirical studies on software evolution is relatively low, despite being a field of research for more than 30 years. Recent research is exploring whether they are applicable to other domains, such as systems developed using eXtreme Programming models, based on the COTS paradigm and on libre software.

A first approach in this field that comprises libre software projects is an article by Burd et al. [Burd & Munro, 1999]. They evaluate the evolution of the GCC, a compiler collection written mainly in C. A different approach is suggested by Fischer et al. in [Fischer *et al.*, 2003; Fischer & Gall, 2004]. Data from the CVS versioning system and from the bug-tracking system is populated in order to manage current and future software evolution. Their approach uncovers hidden dependencies between features and presents them in an easy-to-access visual form. As case study, the Mozilla project was selected.

Software growth

One of the laws that has been more frequently studied is the Fourth Law [Lehman & Belady, 1985; Lehman *et al.*, 1997], which states that the rate of development over the life of a program is approximately constant, and independent of the resources devoted to it. Both Lehman and a statistical study performed by Turski [Turski, 1996] found that those software systems follow an inverse square growth rate. The equation given by Turski is:

$$S_i = S_{i-1} + E/(S_{i-1})^2 \tag{2.1}$$

where S_i is the estimated size of the system at the i-th release (in number of source modules[29]) and E is a parameter. An explanation that is given for this equation states that for a system of size n (modules), the maximal number of possible interconnections is $n \cdot n - 1$. As the system grows, introducing new modules will impact a growing number of existing ones and more effort will be required [Lehman *et al.*, 2001].

When solved directly, the equation is approximately

$$S = (3E \cdot t)^{1/3} \tag{2.2}$$

where S is the size of the system measured in modules, t is time and E a parameter.

The most relevant study on the evolution of libre software projects is probably the one by Godfrey and Tu [Godfrey & Tu, 2000], who studied the Linux kernel in 2000. They found that Linux, then about 2 million lines of code in size, had a super-linear growth rate, apparently in contradiction with Lehman's fourth *law*, and with the statistical evidence from Turski.

The main conclusions of this work can be summarized as follows:

- The Linux kernel exhibits a super-linear growth rate. Most of the growth is due to new functionality and added hardware support, not to bug fixing.

- Much of the functionality (specially device drivers) is complex and extensive, but also relatively independent from each other, and from the rest of the system.

[29] Although there is no precise definition of what a module is as it varies from system to system, in general a module refers to an individual source code file.

- External contributions (both for adding and maintaining code) were frequent in the devices and architecture subsystems. Maintenance is often done by third parties.
- Large parts of the kernel (specially device drivers) do not require active maintenance, but are still shipped with Linux just in case the user needs them.
- The fourth *law* of software evolution is presumably not fulfilled in the case of Linux.

In a later paper, the same authors give the following software growth equation, based on statistical analysis [Godfrey & Tu, 2001]:

$$y = 0.21 \cdot t^2 + 252 \cdot t + 90,055 \tag{2.3}$$

where y is the size in uncommented lines of code and t the days since version 1.0. The coefficient of determination, calculated using least squares, has a value of $r^2 = .997$.

There is also an study by Succi et al. [Succi *et al.*, 2001] about growth in libre software systems, which confirms this super-linearity for the Linux kernel, but finds linear growths for GCC and Apache.

Both studies have been considered by Lehman et al. [Lehman *et al.*, 2001] as anomalies of the *laws* of software evolution, encouraging at the same time for further research on this topic.

More recently, Koch has presented a large-scale software growth analysis of thousands of libre software projects hosted in SourceForge [Koch, 2005]. The author uses a linear and a super-linear quadratic function to fit the growth plots for the software projects under study. Koch states that for his sample a quadratic model performs significantly better than the linear one. He also observes that the growth rate decreases over time in accordance to Lehman's forth *law* and Turski's findings, but that for large libre software projects super-linear growth rate seems to be maintainable. All in all, the detailed methodology that has been used in this study is not clarified in the paper. We do not know if the software growth figures have been taken considering modules (files) as the basic element, source lines of code or just lines of code (LOC) as CVS versioning system provides them. If the last option is the case, then it is not reported if only source code files have been taken into account or other file types (documentation, translation and others) are also considered.

Structural evolution

The relationship between the growth of code components and the evolving code structure had not been empirically studied in detail until Capiluppi et. al looked at one middle-sized libre software application, ARLA[30]), and observed that the various size measures that they apply (LOCs, SLOCs, KbS, files and folders) grow over releases, with close similarity patterns among all of them besides in the case of folders that show different growth trends [Capiluppi *et al.*, 2004b].

Capiluppi assumes the structure to be a proxy of how difficult it is to understand a software architecture. In [Capiluppi *et al.*, 2004a] therefore he studies how a software architecture evolves over time by looking at the directory tree of libre software applications. This idea is worked on with more detail in [Capiluppi, 2004] where source files, source folders and source levels (directories, subdirectories, etc) are identified and tracked over time. The author shows a graphical view of the architecture of the project and compares the different components with the staff that works on it.

The structural evolution of software can also be studied on product families, i.e. software products that are based on the same architecture but that evolve differently because of being adapted to various environments or because they differ in their goals. In this regard Fischer et al. [Fischer *et al.*, 2005] search in the comments that can be attached to the commits to a versioning system for keywords that refer to other members of the family. The three flavors of BSD (FreeBSD, NetBSD and OpenBSD) and Linux offer a good benchmark to find out how big the influence on the others is.

For the same purpose, but following a completely different methodology, Yamamoto et al. measure the similarity of the three BSD operating systems [Yamamoto *et al.*, 2005]. This analysis is done with code clone detection techniques applied to the source code. The authors show that their results match the historical trend of these three projects (OpenBSD being a derivation from NetBSD and NetBSD being a former fork of FreeBSD).

[30] ARLA is a libre software implementation of the AFS, see project website at http://www.stacken.kth.se/projekt/arla/.

Release Management

Release management is concerned with the problem of publishing a software version for the main public. In general, the main goal of releases is to promote the software as much as possible. This is done by providing the software in an easy-to-install manner and to encourage its distribution. The task of delivering releases requires a certain amount of human and technical organization, and practices adopted differ from project to project.

Erenkrantz has proposed a taxonomy for identifying common properties of release management across libre software projects [Erenkrantz, 2003]. The author has studied in depth some large and known projects such as the Linux kernel, the Subversion versioning repository and the Apache web server. The characteristics that have been selected are: (1) the type of release authority (i.e. the person or group who is in charge of the release process), (2) the versioning policy that specifies the version number (version numbers may have special meanings for the development community, as several branches may be developed in parallel), (3) the pre-release criteria that are used (for example, if there are acceptance tests or if no critical bugs are known), (4) who approves that the release is ready to be published, (5) the distribution mechanisms which include the visibility and accessibility of the new versions and (6) the format of the releases (i.e. the type of packages).

2.3.3 Software compilations

Software compilations have been rarely studied in software engineering. This is probably due to the intrinsic difficulties (mostly legal) that software companies have found when integrating large amounts of software programs built by several vendors. The public availability of source code and the possibility of freely redistributing the software avoids this burden in the libre software world. In addition, grouping together software from several authors and making the system easy to install, configure and maintain has provided with a new business opportunity, so that many efforts in this sense have been done in the last years, yielding in the many distributions that we can find today in the market. Distributions occupied, consequently, a space that in the world of proprietary software seldom reaches important proportions: integrators. Their work consists on taking the sources -generally from their original author(s)-, to group them with other tools and applications that could be interesting and to pack everything together in such a way that the task of installing or of updating enormous amounts of packages is easy enough for the common end user.

So, at the beginning of the nineties, the first distributions arose from the union of the GNU tools with the Linux kernel. Its purpose was to facilitate the installation of libre tools as far as possible, an arduous task that requires much patience and knowledge. The second big innovation of distributions -already in the mid-nineties- is due to the package management systems that allowed not only to install a distribution easily, but also offered the possibility to manage (remove, add or update) packages once they had been installed.

A large number of distributions, each one with its own peculiarities, exist. Although the number of distributions can be taken as some thousands, a small set has gained more popularity. Among these, Red Hat and Debian are two of the most known ones. Even having the same purpose, there are considerable differences between Red Hat and Debian; the main one is that Red Hat is a commercial distribution built by the employees of an enterprise, while Debian is built entirely by volunteers. This has several implications on the number of packages they redistribute, the release policy and schedule they follow and on other organizational and management issues.

Red Hat

David A. Wheeler reports the size of the Red Hat 7.1 version in an article entitled *More than a Gigabuck: Estimating GNU/Linux's size* [Wheeler, 2001]. The analysis is based on counting physical lines of source code by means of the SLOCCount tool (see 2.2.3), using the output in SLOCs to feed a Basic COCOMO model [Boehm, 1981]. Wheeler concludes that building Red Hat from scratch using a classical development methodology would suppose a cost of over $1,000 million. Therefore, around 8,000 person-years would be required, although the estimated schedule (which considers all applications developed in parallel and hence gives the amount of time required for building the largest

one) is of 6.53 years. A previous report on Red Hat's 6.2 version threw 4,500 person-years, so a 60% increase from one version to another could be observed [Wheeler, 2000].

Wheeler also gives a detailed analysis of the top-35 largest packages and focuses specially on the largest one: the Linux kernel. The largest subsystem in it is the *drivers* subdirectory, so the reason of the size of Linux is mainly because of the hardware that it supports. Another important subsystem is the *arch* subsystem, again because of a wide range of architectures that are supported by Linux. A later paper on the Linux kernel (version 2.6) uses the Intermediate COCOMO[31] estimation model [Wheeler, 2004]. Wheeler estimates all parameters specified in that model for the development of Linux, considering also Linux as a semi-detached development type, and obtains that it gives a cost increase of around 70% in comparison to the Basic COCOMO model. The figures in this case are $612 millions and over 4500 man-years.

Debian

Debian is probably the largest software collection in history with over 220 million lines of code in its almost 10,000 source packages in the latest release (July 2005), so many find that it can be considered as a proxy for studying the whole libre software phenomenon, at least of the software that is *successful*[32]. González-Barahona et al. have found evidence that the amount of available libre software measured in source lines of code gets doubled around every two years [González-Barahona *et al.*, 2001; 2004]. This means that the code base to be maintained grows steadily over time and that bigger efforts have to be taken into consideration. It can also be interpreted as the available libre software code base doubling every two years.

The evolution of the top 10 largest applications shows how Debian -and the libre software world in general- has shifted from mainly server and administrator tools like operating system kernels, compilers and debuggers to graphical end-user applications in recent years with the inclusion of the Mozilla Internet suite and the OpenOffice.org office suite.

Regarding the programming languages that are used, C is the predominant one with over half of the share of the total pie. But, although increasing in global terms it is losing momentum in favor of other languages like C++, Java and some scripting languages like Python and PHP.

2.3.4 Holistic (ecology) studies

Holistic studies have targeted the whole libre software phenomenon without differencing projects regarding their size or other attributes. This type of studies, not only performed by software engineers but also by social scientists, psychologists, among others, are not only interested in the libre software development strategies, but also on socio-economic factors that drive developers to create and work on projects, in most cases even without direct economic benefits.

SourceForge is one of the most studied software repositories, in part because it is the largest libre software development site that offers free services to those wanting to host their project. Nonetheless, it should be noted that most of the large and known projects have their own development infrastructure and do not use SourceForge services at all. At the current time, it hosts over 100,000 projects. Over 1,000,000 developers are registered in the SourceForge platform. SourceForge is probably a good scenario for studying the libre software ecology, but not the best one for gaining knowledge on *successful* libre software development. For studying successful practices, projects that are usually not hosted at SourceForge should be researched. Among the projects which do not use SourceForge we can find Apache, GNOME, KDE, Linux, Mozilla and many others.

Hahsler and Koch discuss a large-scale data collection methodology for libre software projects based on the assumption that data can be obtained cost-effectively and that even longitudinal data is available to consider the whole project history [Hahsler & Koch, 2005]. Although the methodology

[31] Intermediate COCOMO is an extension to the Basic COCOMO model; it considers cost drivers attributes in addition to the number of SLOC to obtain the final cost estimation.

[32] The interpretation of *success* here is to be included in the Debian distribution, i.e. that a volunteer finds the software useful and has the time to build a software package from the source code.

focuses on the CVS versioning repository, it could be considered with more or less changes for any other way of accessing the sources over time and of measuring the developer involvement in the project.

The methodology is divided into six steps (1) project retrieval and parsing, (2) module retrieval and parsing, (3) status (if CVS available) and parsing, (4) CVS (checkout and log) string generation, (5) CVS string execution and (6) CVS log parsing.

The metrics that the authors propose and discuss are lines of code (LOCs), commits as a proxy of activity, the total time on the project for developers (which is given by the time from the first to the last commit for any developer and could be seen as the upper bound of the time devoted to the project) and a development indicator (as manually introduced by project owners at SourceForge).

The range of analyses is the most interesting contribution:

- Site level analysis: The analysis is done SourceForge-wide, so that developers contributing to more than one project are considered and explored. In the study performed by the authors, they found that almost 95% of the developers worked on three or less projects, which means that the collocation of projects on the same site may not lead to increased participation in other projects.
- Project level: Focused on project-related results such as software growth patterns and correlation of the project size with the number of contributors and the number of commits.
- Participant level: Displays the contribution and the distribution of contributions among participants using Lorenz curves and Gini coefficients.
- Productivity: The number of programmers and the progress within each project over time intervals should be analyzed. To uncover the effects of a higher number of developers working on a project, mean number of commits and of LOCs added per programmer in a period of time should be considered.
- Effort: Effort calculation can be achieved by means of applying the COCOMO I [Boehm, 1981], COCOMO II [Boehm *et al.*, 2000], or the Rayleigh curve [Norden, 1963] models.

But many other researchers have put their eyes on the SourceForge site. For instance, Krishnamurthy analyzed the 100 most-active mature projects hosted there [Krishnamurthy, 2002]. He concluded that most of the studied projects are smaller than those that are widely known and usually matter of study (i.e. Mozilla, Apache...). His findings can be summarized as follow: (1) the vast majority of mature libre software programs are developed by a small number of individuals, (2) very few libre software projects generate a lot of discussion and most projects do not generate too much discussion, (3) projects with more developers tend to be viewed and downloaded more often, (4) the number of developers working on a libre software projects was correlated to the age of the project and (5) a smaller percent of participants were assigned as project administrators in larger groups.

Healy et al. go further beyond and look at all projects that can be found at SourceForge [Healy & Schussman, 2003]. The authors find that there often exists an inequality that can be modeled after a power-law and which is valid for the number of developers, the number of commits to the CVS versioning repository and the number of messages to the forums. In this regard, Hunt et al. have also discussed the common appearance of Pareto distributions in the activity in SourceForge projects [Hunt & Johnson, 2002]. The median number of developers per project is one, while the 95th percentile is 5 developers, which makes not surprising that only a tiny number of projects have more than a dozen developers. Regarding development activity, the median number of CVS commits for all projects at SourceForge is zero and we have to move up to the 75th percentile to find one commit, while the number of commits for 90% of the projects is less than 100 commits. In other words, little or no programming activity takes place in more than half of the projects, or projects are too small in number of developers in order to hold a versioning repository for the project.

An examination of the number of messages and their authors across all forums gives that only projects at the 90th percentile and above have more than two posters. In this sense, von Krogh et al. have noticed that for the libre software project they studied (FreeNet) 1.1% of the population accounts for 50% of the total e-mail list traffic, yielding a Gini coefficient for message authorship of 0.89 (reflecting a high concentration) [von Krogh *et al.*, 2003].

On the other hand, the measures of user interest that Healy and co-authors define (basically site views and software downloads) are closely related to the software activity measured for the projects. Although the most downloaded projects correspond mainly to end-user applications, the most heavily developed projects are mainly system applications that run in the background, programming environments or basic utilities that provide functionality to an operating system [Healy & Schussman, 2003].

2.3.5 Characterization of libre software development

Mockus et al. presented a study comparing the development process of both libre software and non-libre software [Mockus *et al.*, 2002]. Furthermore they selected two different kinds of libre software projects: Apache, which has been community-driven since its birth, and Mozilla, which had a non-free past, but was at the time of the study a community-driven project (although in part still led by a company).

First, the authors give evidence on some differences that exist between libre software and non-libre software projects. The former ones are developed by a huge group of developers, while tasks are not enforced so any developer can choose what to work on. There are no design patterns established and formal planning or rigid scheduling is missing. Hence, the nature of libre software is dispersed both in geographical and administrative terms, while the coordination mechanisms are not very formal.

The paper addresses several questions about libre software projects by studying empirically the Apache project, drawing some initial hypothesis from it and then contrasting the hypothesis with the Mozilla project. The data sources that have been used include the CVS versioning repository, the bug reporting system and the developer mailing list, although mailing list is only considered for Apache.

The findings for Apache throw that there is a core group that is composed of volunteers who are partial time developers. The work to be done is specified in the developer mailing list or the bug reporting system. For the Apache project, 15 developers performed 80% of the atomic commits[33], while 15 developers are responsible for 60% of all bug fixes. In addition there are over 400 developers that have non-punctually contributed to the project (182 fixed 695 bugs, and 249 contributed with 6,092 code submissions). Other 3,060 persons submitted 3,975 bug reports, which is a larger group that contributes only punctually. von Krogh et. al. make the same observation for FreeNet project. Although they observed many participants in the discussions in the mailing lists, writing code is a task that is concentrated in a small set of individuals [von Krogh *et al.*, 2003].

From the study of the developer mailing lists by Lanzara and co-authors of the Apache and Linux projects for two weeks in 2002, some conclusions can be drawn [Lanzara & Morner, 2003]. Few threads are being participated by more than a small number of posters (average number of participants per thread is 2.2 for Linux and 3.1 for Apache). The mean number of messages per thread is also low (3.7 posts per thread for Linux and 5.2 for Apache).

The lifetime for a thread is generally very short: 2.9 days for Linux and 9.7 days for Apache, while the minimum is one day and the maximum achieves almost 100 days, although the latter is an exception as most conversations do not reach more than one day (30% for Linux and 55% for Apache).

Interestingly enough there is a high percentage of first mails that are left unanswered and hence do not build up a thread; this grows as much as to 47% for Linux, while in Apache it is of 19%. A nearer investigation of the posts that contain direct questions throws that 38.75% of them were left unanswered in Linux while 17.1% were in Apache.

In comparison to the non-libre software systems that are considered, Mockus and colleagues find that Apache shows a lower defect density and bugs and problems are fixed earlier. Priority (as perceived by the core group) is respected, so that high priority reports are fixed earlier that less-priority ones [Mockus *et al.*, 2002].

From the study of the Apache web server the following hypothesis have been enunciated:

1. Libre software developments will have a core of developers who control the code base (less than 15 developers do 80% of the contributions).

[33] The authors originally use the term *modification request*.

2. If a project is larger, a strict code ownership policy should be adopted, creating several subprojects.

3. Testers are a group larger by an order of magnitude than the core.

4. If a project has a few main developers, but lacks a large number of contributors, it will not succeed because defects will not be able to be repaired.

5. Defect density is lower in libre software projects than in commercial ones.

6. Libre software developers are also libre software users.

7. Libre software projects can respond faster to customer requests than commercial projects.

In the case of Mozilla, the authors have found that at the beginning there was a lack of external contributors. It was the Mozilla.org staff who controlled development and the work to be done, which was specified in a roadmap. But after some time, external contributions have become more common, although the organization of the project differs from the one found in Apache primarily because of its size (in number of contributors and of the software).

Mozilla counts with 486 developers who have submitted 412 bug fixes, while 6,387 occasional contributors reported 58,000 problems. All in all, only 20 developers perform 80% of the work. Testers are not external and there exists rigid module ownership. The defect density is lower but problems are solved later than in Apache. Priority is respected as well.

All these findings validate the hypotheses previously enunciated for Apache, although the second one was slightly modified to the following wording:

- If the core group has more than 15 developers, formal communication, planning and scheduling protocols should be used.

The final conclusion of the paper is that projects that are not too large do not require over formal communication mechanisms, neither referred to planning and scheduling issues. A corollary is that large projects should be divided into subprojects, inducing to have some territoriality over a module or subproject by a member of the core group. Modularity (i.e. module independence) is vital, and libre software projects have been found to be more modular than the non-libre projects considered in this study. Finally, the defect density reported is lower and the time that it takes to solve defects is smaller in libre software projects. In the author's opinion, modularity is a key issue for this behavior.

Dinh-trong et al. have replicated the study on the FreeBSD project and have obtained similar results and conclusions [Dinh-Trong & Bieman, 2004; 2005]. The first and second hypothesis proposed by Mockus et al. are also supported in the GNOME project [Germán, 2004b].

Slightly different results can by found in an article by Paulson et al. [Paulson *et al.*, 2004]. They perform an empirical study which compares libre and non-libre software projects using five software metrics with the intention of investigating if some common assumptions in favor of libre software can be empirically validated or not. The assumptions are that libre software development fosters faster system growth, that it fosters more creativity, that projects succeed because of their simplicity, that they have fewer defects and that they are more modular.

Therefore several well-known and large libre software projects (Apache, Linux and GCC) are selected and compared to three non-libre projects from the embedded wireless world domain. The results throw that project growth is linear (in number of functions and lines of code added) over time for libre software projects, while the development periods are similar for both kinds of software types. Regarding creativity, the authors observe a linear approximation of growth rate (which is cyclical), so libre software projects seem to foster more. On the other hand, libre software projects are more complex, but show fewer defects and if there are defects, they are found and fixed earlier. Finally, modularity is found to be given more frequently in non-libre software projects. The biggest drawback of this study is that it lacks critical mass to be statistically valid: comparing three randomly-taken libre software projects with three projects from a specific domain and drawing conclusions is risky.

Another detailed field study of a large libre software project is the one by Koch & Schneider [Koch & Schneider, 2002]. They present a methodology for software engineering research into libre software projects using data retrieved from a publicly available CVS repository and mailing lists. This methodology is then applied to the GNOME project and results concerning the programmers and files constituting this development effort together with the progression of the project over time are described. The authors use several metrics in order to study GNOME:

- LOC (Lines-of-Code) i.e. any type of line including comments and blank lines. Gives an idea of the changes in size by subtracting LOCs deleted from LOCs added. Again, a majority of programmers contribute with a small amount of LOCs.
- CVS commits, defined as a "submission of a single file by a single programmer". A majority of programmers also contribute with a small amount of commits, but here inequality among developers is much smaller than in the case of contributing LOCs.
- Time spent by programmers in the project, which is given by the "difference between the first and the last commit" for each developer. This is, of course, much more than the time that the developer has really spent working on the project. The grant total measured can be normalized with the amount of time developers devote to the development of libre software (this data can be obtained from surveys).
- Time spent on each file, defined as the "time between first and last commit" for a given file. As for the previous metric, this can be understood as an upper bound.
- Posts to mailing lists. The histogram of mail posts is similar to the one with commits, so there exists also a high inequality. More productive programmers also are more active in mailing lists.
- Finally, a set of derived metrics is proposed: LOCs added per commit, LOCs added per hour, among others.

Finally the authors use some classical effort estimation models that derive from Norden (modeled as a Rayleigh curve) [Norden, 1963] and Putnam [Putnam, 1978] to see how much developing GNOME may have cost: the results are 169.9 person-years. The model assumes that once given the peak point of the effort, we may infer the total cost of the project.

2.3.6 Fine-grained analyses

Germán has authored several studies that try to gain knowledge on the libre software development process by analyzing the publicly available data from CVS repositories at a fine-grained level, using the groupware suite Evolution as a case study [Germán, 2004a; 2004e].

First, the nature of atomic commits (AC)[34] is considered, defining code atomic commits (codeACs) as the ones that include source code files. These atomic commits are ordered in different types (the list is incomplete and items have not to be mutually exclusive): maintenance (defect fixing), functionality improvement, documentation, architectural evolution (major changes in API, etc.), relocating code and branch-merging. Among code atomic commits, two different types are identified: bugACs, corresponding to a bug reported in BugZilla, and commentACs, which are codeACs where only comments have been *touched*.

Germán observes that most atomic commits contain very few files. The most frequent case is an atomic commit containing two files. This is because GNOME uses changelog files that are committed simultaneously with the files that are changed. Atomic commits fixing defects usually contain few files while documentation atomic commits tend to contain more files than other type of atomic commits.

The author investigates if the hypothesis by Fischer and Gall [Gall *et al.*, 1997; Fischer *et al.*, 2003] that says that historical modifications logs can be used to detect a coupling relationship between two files. If two files are modified at the same time, then these two files are related. Three metrics are therefore defined: (1) modification coupling is the likelihood that if file A is modified, B is also

[34]The author originally uses the term *modification requests* for *atomic commits*, the preferred one in this thesis

modified, (2) frequency of modification, Frequency(A,B), gives the number of atomic commits that contain both A and B and (3) modification neighbors, neighbors(A), is the set of files that have been modified in the same atomic commit as A. After some cleaning (codeACs that include first versions or branches are removed, among others) files that are committed together arise. As expected, the author finds also coupling between .c source code and .h header files.

Further, following hypotheses are verified: (1) a file tends to be modified with the same files and (2) most atomic commits are composed of files that belong to the same module. To test these hypothesis a coupling graph was created, depicting modification coupling in a given period, a sort of (social) network analysis but with files as nodes and edges as couplings. The results yield that the maintenance period had fewer atomic commits than the improvement period. In addition, graphs tend to be composed of small disconnected subgraphs, or clusters of nodes, interconnected by few edges. Another finding is that the modularization of the project has a profound impact in the disjointness of the different subgraphs.

The author shows how programmers are related to modification coupling. The observations that he makes are that most files are modified (*owned*) by one individual developer and that there exists a high correlation between the perceived core maintainer of a module and the most connected programmer to that module in the graph. Finally, the author is also interested in knowing the evolution of functions or method changes in a file over time. The results show that bugACs tend to add and remove less functions than atomic commits in general.

2.4 Socio-cultural and organizational studies

Software development is a human-intensive task. The way human resources are managed and how the development teams are structured has been an ample field of research since the early days of software engineering [Weinberg, 1998; Frederick P. Brooks, 1978] and is still a topic of many efforts in recent years with the proliferation of *agile* methodologies [Beck, 1998] and the libre software phenomenon. The main differences between the environment where libre software is developed and traditional ones have been fostered by philosophical, by technological, by pragmatical and surely by sociological reasons, too.

Sharing knowledge is the main philosophical reason that is behind the development of libre software and that is argued mostly by those who claim that libre software is an ethical question, usually lead ideologically by the Free Software Foundation and other organizations [Stallman, 1999]. The analogy with the scientific world where data, results and knowledge are disseminated among research groups for the benefit of all human beings are key points for this attitude [Bezroukov, 1997].

Technologically, the rise of the Internet in the last decade has made it possible to cooperate with others in creating a software in an easy and an cost-effective manner. Interest groups around any topic have been formed on the Internet and being computer scientists and software developers specially familiar with the Internet environment, they have been precursors in this sense. Global (or distributed) software development is a topic that has rosen much interest among the software engineering community recently [Herbsleb *et al.*, 2001]; many libre software projects, if not all the ones that have an ample community, rely on this way of creating the software [Germán, 2004b].

As distributed development has shown to be efficient, pragmatic points of view have moved towards the idea of introducing libre software practices. These reasons have been more recently enunciated and are heavily based on the possibilities that the Internet offers. The idea behind this point is very simple: if somebody is willing to contribute to a project, this should be allowed and boosted as this is beneficial for the project itself. This point relies on the technological advances that have been pointed out above, as well as on the benefits that sharing knowledge brings and is the standpoint of groups like the Open Source Initiative. In any case, it should be noted that although tightly bound, pragmatists (those who try to achieve the best result by sharing knowledge) and ideologists (those who see sharing knowledge as the main cause in itself) position themselves in different groups, the former around the Open Source Initiative while the latter's main player is the Free Software Foundation.

Finally, there are sociological reasons that have been important for the development of libre software. So, for instance, the spread of English in the last decades and its use as *lingua franca* in the

technical world have made it possible that libre software development can take part in a distributed way. Of course, many other reasons may fall in this field. There has not been much empirical work in this area, so that many of these questions are yet unresolved, though some research projects have started to concentrate on them (as for instance, the EU-funded FLOSSPols[35] and FLOSSWorld[36] projects).

In the remainder of this section we will present related research that has been performed on human resources in libre software development. Beyond the reasons that have been presented so far, it will be centered on management, organizational and procedural aspects. The point of view will be software engineering related, which means that even if other interpretations are possible, and of course are made by other research groups from other research fields, we will try to stick to aspects that influence the software engineering process.

2.4.1 Organizational structure of libre software projects

The *human* factor becomes especially interesting for two reasons. First, attending to Conway's *law*, organizations that design systems are constrained to produce designs which are copies of their communication structures [Conway, 1968]. And second, it may be used to infer the development and maintenance costs of the software development life cycle. Some works have already studied the latter aspect, trying to figure out, by means of studying software repositories, how much effort is to be applied when changing a software system [Graves & Mockus, 1998].

Lanzara et al. characterize the process of knowledge generation in libre software projects as an ecology of agents, rules, activities, practices and interactions [Lanzara & Morner, 2003]. In the authors words, "this field supposes a rich field to expose the creation, accumulation and dissemination of knowledge in distributed teams in a fast and cost-effective manner even in situations where membership is not tightening and participation is volatile".

Although some of the self-organizing properties of interactive systems can be identified in libre software projects, there are also some characteristics that are common of formal organizations. Thus, in addition to fluid and informal participation, there exists also stable membership. On the other hand, governance mechanisms co-exist with largely non-governed processes. Another interesting characteristic is that libre software projects are able to build up memory differently from pure interactive systems and have representation mechanisms which allow them to interact with their environment. But the most interesting fact is that all the organizational gear is implemented through technological means, so the technology has to be investigated in order to find the organizational structure.

Management in large libre software projects

Germán gives a detailed description of the inner functioning of GNOME, a large libre software project regarding its management, decision and governance structure with several companies involved in the development and thousands of contributors worldwide [Germán, 2004b]. The purpose of the study is to qualitatively analyze how the factors that compose Global Software Development (GSD) -mainly distance, time differences and cultural differences- affect the development of libre software and how GNOME faces these problems and has found solutions.

First, Germán describes how the GNOME source code base is organized. Given the magnitude of the project, it is divided into modules of which he identifies four types: (a) required libraries, (b) core applications, (c) applications and (d) other. Each module has its own maintainer, set of developers and development time-line and goals. Although modules are interrelated, relationships are tried to kept minimal, so they can evolve independently.

Then, management and direction is studied. It is pointed out that the mere presence of a widely-known developer, Miguel de Icaza[37], among the libre software community in the beginnings of the GNOME project has been a key factor for the early success of the project. But leadership has become

[35] Free/Libre/Open Source Software: Policy Support (acronym FLOSSPols): http://flosspols.org.

[36] Free/Libre/Open Source Software: Worldwide impact study (acronym FLOSSWorld): http://www.flossworld.org.

[37] Miguel de Icaza is co-founder of the GNOME project and a recognized figure in the libre software wold in general.

more complex with the increasing number of modules and developers that have entered the project since then. That is the reason why in the year 2000 the GNOME Foundation was founded. The GNOME Foundation is composed of four entities: (a) members, which can be any contributor to a project being contributions of any kind (from source code submission to documentation), (b) a board of annually elected directors that is in care of administrative and other tasks, (c) and advisory board which is composed of corporate and non-profit organizations and finally (d) an executive director, a paid employee of the foundation and who helps with administrative and other tasks.

The GNOME Foundation describes itself as a meritocratic democracy where contributors to the project are the *demos*. Contributors are usually coders, but not only. Germán identifies several types of them: (a) paid employees who usually fulfill tasks that are less attractive to volunteers, as for instance building and maintaining the software infrastructure of the project (servers, mailing lists, bug tracking system, CVS versioning system, among other), (b) volunteers who devote their spare time in working on GNOME (interestingly enough many of the paid developers in GNOME were at some point in the past volunteers, so their hobby has become their job) and (c) non-programmers, including documenters, translators, designers and other project-related activities such as press contacts.

Germán adds a final group of newcomers and states that even if there is no detailed description or data about this issue, the cost of entry has increased over time due to the rising organizational and technological complexity of the project. The GNOME project seems to be aware of this problem and has fostered some ways of attracting new collaborators. The initiatives include: (a) to facilitate bug fixing by means of bug-squad days, (b) to have TODO lists with minor tasks and (c) to use the GNOME-love mailing list where newcomers may be attended and advised.

For some specific tasks, committees are created. Such is the case for fund-raising, the organization of an annual conference or the release team. The main communication mechanisms are the mailing lists, but the use of Internet Relay Chat, web sites, the annual conference and weekly summaries are also of great importance for the dissemination of information around the project.

In the authors words, the conclusions that can be learned from the GNOME experience for Global Software Development projects are (a) to flatten the organization and allow more participation in decision-making, (b) to use multiple types of communication, (c) to treat developers as partners in the development process, (d) to have a regularly scheduled co-located meeting where the main decisions can be made, (e) to create task forces that work aside the different teams, (f) to set clear procedures and policies for conflict resolution, (g) to modularize the product in order to minimize communication and (h) to ask developers to document their daily contributions.

Brand has made a similar study on the KDE project [Brand, 2004]. KDE shares with GNOME the goal of building a *universal* desktop environment and has a community that is similar in size to GNOME. Besides pointing out how the organization of the KDE project is, who is in charge of making decisions and how conflicts are resolved, Brand as a sociologist shows also interest in the motivations of developers and how all this is achieved by means of a virtual working environment.

The onion model

But, probably the most well known model about the organizational structure of libre software projects is given by the *onion model* [Crowston *et al.*, 2003a; Crowston & Howison, 2003; 2005]. The *onion model* is a visual analogy that tries to represent how developers and users are positioned in communities. In this model, as shown in figure 2.2, the authors differentiate between core developers (those who have a high involvement in the project), co-developers (with punctual, but frequent contributions), active users (that occasionally contribute) and finally passive users. At a finer level of detail, we could consider specific roles; so, for instance, we could have the project founder (initiator) or a release coordinator in charge of the release management (see subsection 2.3.2 for further details on the release management procedures).

Some studies have already reported and quantified this structure for several libre software projects ([Mockus *et al.*, 2002] for the Apache web server and the Mozilla web browser, [Dinh-Trong & Bieman, 2004; 2005] for the FreeBSD project). According to these studies the core is composed of a small group of about a dozen developers. Surrounding the core group there is a group of contributors, about an order of magnitude larger, who send bug fixes and eventually submit some code. Still an order of

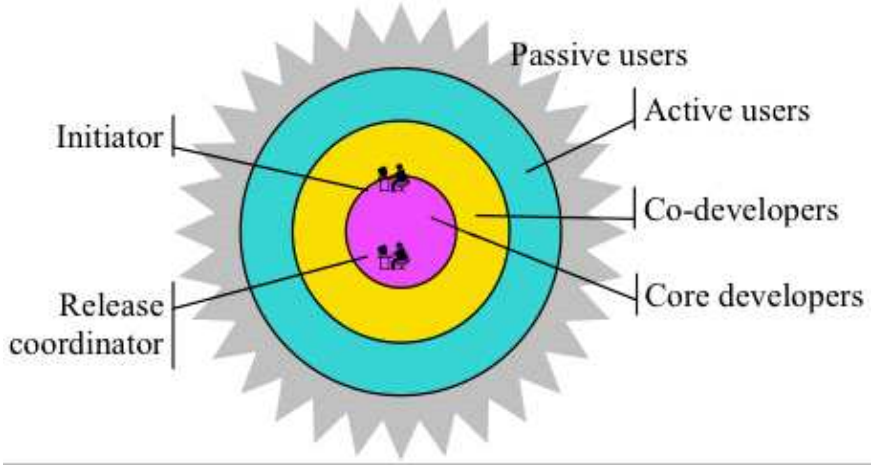


Figure 2.2: *A synthesized libre software development team structure, also known as the* onion model. *Source: [Crowston & Howison, 2005]*

magnitude larger is the number of casual contributors who occasionally report bugs or perform other small tasks. Finally, we find the surrounding user community which may serve as a pool for future developers or casual contributors. The extension of the user community is difficult to account in quantitative terms. The reason for this is that software can be redistributed by third parties (as for instance, the many distributions, from other web sites, from friends, among others), so usually the number of downloads of the *official* project site is not a good indicator of its use.

Monitoring volunteers

One of the most astonishing characteristics of libre software development is its heavy dependency on contributions done by volunteers. Of course, coordination mechanisms in use in libre software projects have to face both the high number of volunteers and the geographical distribution of participants.

Michlmayr has studied how to manage the volunteer activity in the Debian project [Michlmayr, 2004], a project composed of over 1,000 volunteers (called Debian Developers). The Debian Developers are special compared to what commonly can be assumed to be a volunteer in other libre software projects as an admission process has to be surpassed in order to become one. The starting point of this research is that although many volunteers do not play a crucial role in the project, some of them do. In that case, a cease in their involvement may cause the project severe problems, ranging from a lose in the quality to other organizational issues. The author reports two approaches that exist in the Debian project about responsibility: a first group that argues that volunteers are totally free to start and stop collaborating on the project anytime, while a second argues that volunteers should fulfill the requirements they have previously agreed on. In both cases monitoring volunteers is hardened by the fact of having a distributed environment.

In the case of Debian, once a contributor has concluded with success the admission process and becomes a Debian Developer, he usually maintains some software packages. If Debian Developers go on holidays or retire from the project they should inform the project following the Debian Developer's Reference[38]. The Debian project has introduced some infrastructure if this is not the case. So, a tool called *Echelon* monitors the Debian mailing lists for postings from Debian Developers and keeps track their activity. Other information sources of lack of maintainer activity are unmaintained packages (for instance, with unresolved critical bugs for a long time), old standard-versions or new *upstream* versions[39] pending for a long time.

Having an admission process is a characteristic that makes the Debian project special. In other projects, usually contributors have to send their patches to the main contributors who review and

[38] http://www.debian.org/doc/developers-reference/

[39] *upstream* is how the original software is called in the Debian jargon. The *upstream* sources are modified by maintainers in order to create a ready-to-install package that meets the Debian packaging guidelines.

introduce them into the current source code base, serving thus as *gatekeepers* [O'Mahoney & Ferraro, 2004].

2.4.2 Social Network Analysis

A systematic approach to the organizational structure of communities of any kind is given by the techniques used for the study and characterization of complex systems. Special attention has been paid recently to complex networks, where graph and network analysis play an important role that is gaining popularity due to its intrinsic power to reduce a system to its single components and relationships. Network characterization is widely used in many scientific and technological disciplines, such as neurobiology [Watts & Strogatz, 1998], computer networks [Albert *et al.*, 2000; Cancho & Sole, 2001] and linguistics [Kumar *et al.*, 2002]. The techniques can be applied to social relationships among the members of a group and may help to identify the structure of an organization.

In this context, social networks analysis (SNA) is the result of the confluence of several disciplines, including social networks theory, organizational behavior, interpersonal communications, chaos theory, complex adaptive systems, artificial intelligence, operational research, and graph theory. In essence, SNA provides an object-oriented model of a structure. Objects in the model are linked in a complex pattern which shows their topology of interaction, relationship and information flow.

The first problem to solve when using SNA is getting information to construct the network to analyze. One specially interesting kind of data sources are the records maintained by many computer-based systems. As information flow in libre software projects happen through telematic means and, in general, these exchanges are stored and publicly accessible, we can use these data sources to establish the interconnections among members.

Thus, information taken from the data sources available for libre software projects offer information about who, when and how collaboration takes place. These records can be seen as an accurate and detailed picture of the organizational structure of the software and the developers working on it. When two developers work on the same program, they have to exchange in a direct or indirect manner information and knowledge to coordinate their actions, hence building a link or relationship among them. It would be reasonable to think that a relationship is stronger as the number of contributions to a common program is also higher, giving an idea of what could be considered as a weight.

A simple approach in this regard is the one offered by Mutton, which describes a simple network analysis performed with the data obtained from the communication exchange between developers [Mutton, 2003]. The data has been obtained from the conversations held in IRC channels with the PieSpy tool (see 2.2.3). The most interesting part of this work is how social networks are inferred, as it is done by a set of heuristics which are properly described. These heuristics include direct addressing of users (for instance when in a conversation one part specifies the nickname of the other part), temporal proximity so that after a long silence in a channel two users exchange messages or temporal density by two users originating many messages in a short time interval. All this happens in publicly available chat rooms, but IRC also offers the possibility of exchanging private messages among users, which the author discusses. In his words although obtaining these data could be possible if access to the IRC servers is granted, it raises ethical questions and has thus not been implemented.

Affiliation networks

A more sophisticated approach is the one that considers a specific kind of social network which is called affiliation network. These networks are characterized by showing two types of vertices: *actors* and *groups*. Representing the network with actors as vertices implies that a link between two actors is given if they belong to the same group. If we take groups as vertices, then two groups are connected when there is at least one actor that belongs at the same time to both groups. In the case of libre software development, developers take the role of actors and software artifacts (in any fashion, ranging from projects as defined in SourceForge or as modules, directories or even files) are considered as groups. Hence, the *belong to* characteristic has to be reworded to *has contributed to* for our purposes.

This approach will result in a dual view of the same organization: as a network of modules linked by common developers, and as a network of developers linked by common software artifacts

that developers have worked on. Similar approaches have been used for analyzing other complex organizations, like the known network of movie actors [Albert & Barabasi, 2002], where groups are built based on movies where actors have performed together or the network of scientific authors [Newman, 2001a; 2001b] where authors play the role of actors and groups are given by the research papers they have co-authored.

In the case of libre software, social network analysis can be applied in multiple ways. So, for instance, Ohira et al. use social network theory with the aim of accelerating cross-project collaboration [Ohira *et al.*, 2005]. They assume that with the help of such techniques it could be possible to answer questions such as "who do I have to ask?" or "what can I ask?". This is specially important in the case of very fragmented communities with many small projects where knowledge is disseminated and the organizational structure is loose and changes frequently.

But the most known research works have been accomplished by Madey et al. [Madey *et al.*, 2002; 2004]. It provides a basic social network analysis on developers and projects hosted at SourceForge. One big drawback of this analysis is that relationships are considered equal, without considering links stronger when joint work is more frequent. This makes the clustering that the authors perform very poor; they identify the presence of a large cluster consisting of almost 7,000 developers, while the size of the next cluster is 55. The use of weights for the links and the introduction of a threshold would probably give a more realistic picture of the SourceForge community.

Compared to actor-movie networks, where over 90% of the actors belong to one larger cluster [Watts, 1999], the largest developer cluster in SourceForge amounts only 25% of the total developer population. One of the explanations that the authors give is that libre software development is not that well well connected as other social or collaborative networks, although they hint that this could also be due to the maturity of the movie industry in comparison to the libre software development community. An alternative interpretation is that there exists a fragmentation in the technological landscape, because of the effort required to acquire knowledge on a topic.

The most interesting analysis is the one that states that libre software development is not a random graph which would mean that newcomers would attach to existing nodes in a random manner (i.e. with uniform probabilities). What has been observed is a graph that displays preferential attachment of the new nodes in the sense that some existing nodes are more attractive to them. The authors state that this happens typically in situations where positive feedback exists, which in fact gives as consequence the *rich-get-richer* phenomenon (or the *band-wagon* effect). In other words, *success* is a starting point for more *success* as developers show to prefer to contribute to already *successful* projects.

Another interesting observation is the importance of linchpin developers. Linchpin nodes are those nodes that serve as contact point between clusters. They have a strategical position as all information between these two clusters has to go through them. It is then not surprising that Madey et al. say that these developers play a similar role as *gatekeepers* in organizational studies on technology diffusion.

The result that has been obtained shows that preferential attachment is more accurate than random attachment, the latter not giving a power-law behavior. Preferential attachment can be implemented both by taking into account the size of the project or the size of the development team, but it does not model the situation where new projects or developers become quickly linked by many others (the young *start-up* phenomenon), so the authors introduced a concept ("fitness factor") to explain why some web sites have more links despite their recent creation in the web. Adding the fitness factor improves the fit of the model, but some discrepancies also arise. Changes in the fitness factor over time gave better results, meaning that a project may start being very attractive, but this extra attraction is lost over time against the characteristics considered first, such as project and development team size.

Parameters

Once the networks are constructed based on the previous definitions, and the degrees and costs of relationship have been calculated for linked nodes, we can apply standard SNA concepts to define the following parameters of the network (the interpretation of the main implications of each parameter is also offered):

- **Degree**. The degree, k, of a vertex is the number of edges connected to it. In SNA, this parameter reflects the popularity of a vertex, in the sense that most popular vertices are those maintaining the highest number of relationships. More revealing than the degree of single vertices is the distribution degree of the network (the probability of a vertex having a given degree). This is one of the most relevant characterizations because it provides essential information to understand the topology of a network (and if longitudinal data is available, the evolution of the topology). For example, it is well known that a random network follows a Poisson distribution, while a network following a preferential attachment growth model presents a power law distribution [Albert & Barabasi, 2002]. In our context, the degree of a commiter corresponds to the number of other commiters sharing modules with her, while the the degree of a module is the total number of modules with which it shares developers.

- **Weighted degree**. When dealing with weighted networks, the degree of a vertex may be tricky. A vertex with a high degree is not necessarily well connected to the network because all its edges may be weak. On the other hand, a low degree vertex may be strongly attached to the network if all its links are heavy. For this reason the weighted degree of a vertex, k_w, is defined as the sum of the weights of all the edges connected to it. The weighted degree of a vertex can be interpreted as the maximum capacity to receive information of that vertex. It is also related to the effort spent by the vertex in maintaining its relationships.

- **Clustering coefficient** [Watts & Strogatz, 1998]. The clustering coefficient, c, of a vertex measures the transitivity of a network. Given a vertex v in a graph G, it can be defined as the probability that any two neighbors of v are connected (the neighbors of v are those vertices directly connected to v). Hence

$$c(v) = \frac{2E(v)}{k_v(k_v - 1)}, \tag{2.4}$$

  where k_v is the number of neighbors of v and $E(v)$ is the number of edges between them. The intuitive interpretation of the clustering coefficient is somehow subtle. If the total number of neighbors of v is k_v, the maximum number of edges than can exist within that neighborhood is $\frac{k_v(k_v-1)}{2}$. Hence, the clustering coefficient represents the fraction of the number of edges that really are in a neighborhood. Therefore it can be considered as a measurement of the tendency of a given vertex to promote relationships among its neighbors. In a completely random graph, the clustering coefficient is low, because the probability of any two vertices being connected is the same, independently on them sharing a common neighbor. On the other hand, it has been shown that most social networks present significantly high clustering coefficients (for instance, the probability of two persons being friends is not independent from the fact that they share a common friend) [Albert & Barabasi, 2002; Watts, 2003].

  From an organizational point of view, the clustering coefficient helps to identify hot spots of knowledge interchange on dynamic networks. When this parameter is high for a vertex, that vertex is promoting its neighbors to interact with each other. Somehow it is fostering connections among its neighborhood. High clustering coefficients in networks are indicative for *cliques*. Besides, the clustering coefficient is also a measurement of the redundancy of the communication links around a vertex.

- **Weighted clustering coefficient** [Latora & Marchiori, 2003]. The clustering coefficient does not consider the weight of edges. We may refine it by introducing the weighted clustering coefficient, c_w, of a vertex, which is an attempt to generalize the concept of clustering coefficient to weighted networks. Given a vertex v in a weighted graph G it can be defined as:

$$c_w(v) = \sum_{i \neq j \in N_G(v)} w_{ij} \frac{1}{k_v(k_v - 1)}, \tag{2.5}$$

  where $N_G(v)$ is the neighborhood of v in G (the subgraph of all vertices connected to v), w_{ij} is the degree of relationship of the link between neighbor i and neighbor j ($w_{ij} = 0$ if there

are no links), and k_v is the number of neighbors. The weighted clustering coefficient can be interpreted as a measurement of the local efficiency of the network around a particular vertex, because vertices promoting strong interactions among their neighbors will have high values for this parameter. It can also be seen as a measurement of the redundancy of interactions around a vertex.

- **Distance centrality** [Sabidussi, 1996]: The distance centrality of a vertex, D_c, is a measurement of its proximity to the rest. It is sometimes called *closeness centrality* as the higher its value the closer that vertex is (on average) to the others. Given a vertex v and a graph G, it can be defined as:

$$D_c(v) = \frac{1}{\sum_{t \in G} d_G(v,t)}, \tag{2.6}$$

  where $d_G(v,t)$ is the minimum distance from vertex v to vertex t (i.e. the sum of the costs of relationship of all edges in the shortest path from v to t). The distance centrality can be interpreted as a measurement of the influence of a vertex in a graph because the higher its value, the easier for that vertex to spread information through that network. Observe that when a given vertex is *far* from the others, it has a low degree of relationship (i.e. a high cost of relationship) with the rest. So, the term $\sum_{t \in G} d_G(v,t)$ will increase, meaning that it does not occupy a central position in the network. In that case, the distance centrality will be low.

  Research has shown that employees who are central in networks learn faster, perform better and are more committed to the organization. These employees are also less likely to turn over. Besides, from the point of view of information propagation, vertices with high centrality are like *hills* on the plain, in the sense that any knowledge is put on them is rapidly seen by the rest and spreads easily to the rest of the organization.

- **Betweenness centrality** [Freeman, 1977; Anthonisse, 1971]: The betweenness centrality of a vertex, B_c, is a measurement of the number of shortest paths traversing that particular vertex. Given a vertex v and a graph G, it can be defined as:

$$B_c(v) = \sum_{s \neq v \neq t / in G} \frac{\sigma_{st}(v)}{\sigma_{st}}, \tag{2.7}$$

  where $\sigma_{st}(v)$ is the number of shortest paths from s to t going through v, and σ_{st} is the total number of shortest paths between s and t. The betweenness centrality of a vertex can be interpreted as a measurement of the information control that it can perform on a graph, in the sense that vertices with a high value are intermediate nodes for the communication of the rest. In our context, given that we have weighted networks, multiple shortest paths between any pair of vertices are highly improbable. So, the term $\frac{\sigma_{st}(v)}{\sigma_{st}}$ takes usually only two values: 1, if the shortest path between s and t goes through v, or 0 otherwise. So, the betweenness centrality is just a measurement of the number of shortest paths traversing a given vertex.

  In the SNA literature vertices with high betweenness centrality are known to cover *structural holes*. That is, those vertices glue together parts of the organization that would be otherwise far away from each other. They receive a diverse combination of information available to no one else in the network and have therefore a higher probability of being involved in the knowledge generation processes.

High values of the clustering coefficient are usually a symptom of *small world* behavior. The small world behavior of a network can be analyzed by comparing it with an equivalent (in number of vertices and edges) random network. When a network has a diameter (or average distance among vertices) similar to its random counterpart but, at the same time, has a higher average clustering coefficient, we say it is a small world. It is well known that small world networks are those optimizing the short and long term information flow efficiency [Watts, 2003]. Those networks are also especially well adapted to solve the problem of searching knowledge through their vertices.

Table 2.1 summarizes the various SNA parameters, their meanings and the information they provide, that have been presented in this section. These parameters, and their distributions and correlations may characterize the corresponding networks.

Parameter	Meaning	Interpretation
Degree of relationship	Common activity among two entities (measured in commits)	How strong the relationship is
Cost of relationship	Inverse of the degree of relationship	Cost of reaching one vertex from the other
Degree	Number of vertices connected to a node	Popularity of a vertex
Distribution degree	Probability of a vertex having a given degree	Topology of the network (Poisson or power law distributions)
Weighted degree	Degree considering weights of the links among vertices	Maximum capacity to receive information for a vertex. Effort in maintaining the relationships
Clustering coefficient	Fraction of the total number of edges that could exist for a given vertex that really exist	Transitivity of a network: tendency of a vertex to promote relationships among its neighbors. Helps identifying hot spots of knowledge interchange in dynamic networks
Weighted clustering coefficient	Generalization of the clustering coefficient concept to weighted networks	Local efficiency of the network around a vertex. Redundancy of interactions around a vertex
Distance centrality	Measurement of the proximity of a vertex to the rest	Influence of a vertex in a graph. The higher the value the easier it is for the vertex to spread information through the network
Betweenness centrality	Number of shortest paths traversing a vertex	Measurement of the information control. Higher values give that the vertex is an intermediate node for the communication of the rest. Vertices with high values are known to cover *structural holes*
Small world	Diameter (or avg distance among vertices) similar but higher avg clustering coefficient than random network	Optimizes short and long term information flow efficiency. Especially well adapted to solve the problem of searching knowledge through their vertices

Table 2.1: *Summary of the SNA parameters, their meaning and their interpretation.*

2.4.3 Surveys

The information that can be obtained by mining software repositories is by far not complete, even if we track data from various sources. For instance, activity patterns from versioning repositories or mailing lists suppose just punctual interactions without giving detail of how much time they have supposed to the developer who did that task. It does also not give information about how these actions affect the rest of the community, i.e. how much time they devote to read a mail or to check the code that has been just committed. Other interesting (and mainly social) characteristics regarding developers as their age, their civil status or if they benefit economically from their commitment to a project are difficult if not impossible to gather from the data sources that have been presented so far in this thesis. Surveys that target developers fill this gap providing with additional information.

One of the first surveys on developers, if not the first one, has been the WIDI[40] survey in 2001

[40]WIDI is the acronym for *Who Is Doing It?* which explicitly shows the intention of the survey. The on-line results of the survey may be visited at the following location: http://widi.berlios.de.

and which counted with the participation of the author of this thesis [Robles *et al.*, 2001]. Despite the fact that the WIDI survey was performed by computer science students without the valuable help of sociologists and psychologists, its methodology has been followed by many other surveys on libre software developers that have been done to the moment. Basically, it consisted on a web-form with a set of questions. Every question usually had a limited number of choices that could be selected as answers. This was done in order to lower the need of computation and provide immediate feedback to the one who has filled it out. Respondents were attracted by means of posts in libre software news sites and development mailing lists, so their nature can be assumed to be self-selected. This raises some questions about the significance of the sample and how representative it is. Later surveys as FLOSS [Ghosh *et al.*, 2002a] or FLOSS-US [David *et al.*, 2003] therefore asked for additional information that helped verifying that the ones who had filled out the survey were actually libre software developers [Ghosh *et al.*, 2002c]. This was done in the FLOSS survey by asking the (whole or partial) e-mail address and by checking the e-mail addresses against the data obtained from scanning the source code for authorship information with CODD (see section 3.2.5) for an ample range of libre software projects.

In any case, data that is usually acquired through surveys can be divided into several sets. Personal data includes nationality, mother tongue and other spoken languages, gender and age. Questions regarding nationality were used in WIDI to infer the amount of libre software developers for any given country and region (both in absolute terms as well as per-capita numbers), although the self-selection of the sample and the diffusion of the announcement skewed results that much that it is difficult to state if they are significant or not. This is the reason why later surveys do not target to answer that question. Other issues that can be handled in surveys are professional-related facts, age, gender, economic status, education and developer motivation.

A similar kind of survey, but with a slightly different methodology has been used by the Boston Consulting Group - Open Source Technology Group[41]. The authors selected a sample of developers in SourceForge that met certain criteria and sent them an inquiry via e-mail[42]. Their results reveal that the libre software community is composed of participants from many countries (they have found respondents from 35 countries). Respondents are highly skilled IT professionals with more than 10 years of experience on average in the field that observe an extremely high level of creativity in the projects they are involved in. For them having fun, acquiring new skills, the possibility of accessing to the source code and personal needs in the software drive them to contribute in such projects. Contrary to common assumption, developers do not rate defeating proprietary software companies as a major motivator. The amount of time they devote to the development of libre software is in mean 10 hours per week.

Other significant surveys that can be found in the libre software literature are the ones performed on specific projects. These surveys may include interviews and more specific questions and are usually focused on very specific questions. Hertel et al. performed a survey on the motivations of Linux kernel developers [Hertel *et al.*, 2003], while Brand has taken the KDE project to research the recruitment and organization of teams of volunteers on the Internet [Brand, 2004]. Finally, von Krogh et al. have performed telephonic interviews while studying the joining process of members into the FreeNet community [von Krogh *et al.*, 2003].

2.4.4 Joining processes and simulation models

In the following section, we will introduce some related research on organizational processes that occur in libre software environments. It should be noted that in spite of technical development processes [Vixie, 1999], we will focus on social processes, such as knowledge creation or developer integration.

The onion model presented in subsection 2.4.1 gives a static picture of a project, so it lacks of the time axis that is required for studying the joining processes. In this regard, the model has been complemented by Ye et al. with a more theoretical identification and description of the roles in the

[41] http://www.ostg.com/bcg/

[42] In reality, they had two different samples as they specified criteria for two different groups.

model which has added dynamism [Ye *et al.*, 2004]. According to this idea, a core developer is supposed to go through all those roles, starting as a user, until she eventually reaches the core.

Jensen and Scacchi [Jensen & Scacchi, 2005] have also studied and modeled the processes of role migration for some libre software communities, focusing in the case of end-users who become developers for some projects selected as case studies. They have found different paths for the same process, and concluded that the organizational structure of the studied libre software projects are very dynamic in comparison to traditional software development organizations. In comparison to the traditional *onion* model described in subsection 2.4.1, they have identified a richer set of roles (for instance, *code sheriff* in the Mozilla community) that even include marketing and governance issues.

Fichman and Kemerer found that in traditional proprietary software development environments, the barrier of entry for new contributions by users and developers grows with the complexity of the system [Fichman & Kemerer, 1997]. Joining a project may be too costly for newcomers if the system is too complex and only actively involved participants for a certain period of time may understand the underlying software and be able to contribute. von Krogh et al. have studied in a case study how the joining process is for a libre software project [von Krogh *et al.*, 2003] . They chose the FreeNet project because of its innovative characteristics[43]. The study was done in depth and included several data sources, among others telephone interviews with eight developers, the mailing list archives, the logs of the versioning system as well as documents that were publicly available on the Internet site.

Von Krogh et al. have observed the integration process and described a joining script for a libre software project as they have found that although such a process is not previously determined, some common rules apply [von Krogh *et al.*, 2003]. The authors define a *joining script* which has to be followed by a new member entering the community. For that reason, they classify the members of the project mailing list into four groups: (1) list participants, i.e. those who are only active in the mailing list (296 out of the 326 participants), (2) joiners, who are those who have submitted some code but do not have write access granted to the CVS versioning repository, (3) newcomers, who just have begun to make changes to the CVS and finally (4) developers, who have had CVS access for a longer period of time.

The joining script specifies the level and type of activities that joiners have to go through to become developers. Messages to the mailing list are taken as a proxy for activity on the project. The authors find that following the joining script raises the probability of a joiner being granted access to the community. The limitation of this research is that it may not be valid for other libre software projects as it is based on a single case study; i.e. the *joining script* should be validated with other projects.

The integration of new developers in software projects has been matter of several simulation models by several research groups. For instance, based on data obtained from social network analysis (see subsection 2.4.2), Madey and co-authors developed a model [Madey *et al.*, 2004] to better understand the social process. It is an iterative model, where authors are conceptualized as agents. The time period for every iteration is one day. In every period a developer may (1) start a new project, (2) join an existing project or (3) quit a project. Every turn also new developers may join the community. The data collected from SourceForge is used to refine the simulation process.

A similar, and possibly more complete approach, is given by Dalle and David [Dalle & David, 2003]. By means of a stochastic simulation the dynamic and decentralized low-level decisions present in libre software projects are modelled. This model is based upon the idea that reward is the key factor that governs the distribution of developers among the projects. Another simulation model by Antionades et al has, in addition to the elements considered by Dalle and David, attempted to reproduce some quantitative outputs for the libre software projects under simulation (such as lines of code, defect density, and number of participants) [Antoniades *et al.*, 2004].

2.5 Data sources used in research

In this chapter we have seen the various data sources that are generally publicly available from libre software projects and that have been used in literature. To sum up what we have seen, we have made

[43]The FreeNet project is the result of a PhD thesis.

a classification of the papers presented in this chapter. The classification can be observed in table 2.2. We have classified under *other* all those data sources and meta-data which are usually project-specific. In this, sense we have two papers based on the analysis of the changelogs (files that serve as a diary of the changes that have been performed on the software in inverse chronological order), text, entries in the Linux Software Map (meta-data that gives a brief description and some other information about a software package as author, license, version, etc.) and Internet Relay Chat communications.

Data source	Research papers and studies
Source Code	[Antoniol *et al.*, 2005; Burd & Munro, 1999; Ghosh *et al.*, 2002b; Ghosh & Prakash, 2000; Godfrey & Tu, 2000; Koch, 2005; Mockus *et al.*, 2002; Dinh-Trong & Bieman, 2004; 2005; Samoladas *et al.*, 2004; Schach *et al.*, 2002; Succi *et al.*, 2001; Paulson *et al.*, 2004; Wheeler, 2001; González-Barahona *et al.*, 2001; 2004; Ferenc *et al.*, 2004; Capiluppi, 2004; Capiluppi *et al.*, 2004b; 2004a; Godfrey & Tu, 2001; Yamamoto *et al.*, 2005; Hahsler, 2004]
Bug-tracking systems	[Antoniol *et al.*, 2005; Crowston & Howison, 2003; Fischer *et al.*, 2003; Fischer & Gall, 2004; Germán, 2004c; Germán & Hindle, 2005; Mockus *et al.*, 2002; Dinh-Trong & Bieman, 2004; 2005; Paulson *et al.*, 2004]
Versioning systems	[Antoniol *et al.*, 2005; Fischer *et al.*, 2003; Fischer & Gall, 2004; Germán, 2004b; 2004a; 2004c; 2004e; Koch & Schneider, 2002; Hahsler & Koch, 2005; Massey, 2005; Fischer *et al.*, 2005]
Mailing lists	[Koch & Schneider, 2002; von Krogh *et al.*, 2003; Germán, 2004e]
SourceForge	[Healy & Schussman, 2003; Madey *et al.*, 2002; Conklin *et al.*, 2005; Krishnamurthy, 2002; Howison & Crowston, 2004; Hunt & Johnson, 2002]
Developer surveys	[Brand, 2004; David *et al.*, 2003; Ghosh *et al.*, 2002a; Robles *et al.*, 2001; Hertel *et al.*, 2003]
Other	[Chen *et al.*, 2004; Capiluppi *et al.*, 2003; Tuomi, 2004] (changelogs), [Dekhtyar *et al.*, 2004] (text), [Dempsey *et al.*, 1999] (LSM), [Mutton, 2003] (IRC)

Table 2.2: *Empirical research papers and studies classified by data source.*

Based on this classification, we have built a pie that shows the distribution of the data sources in literature (see figure 2.3). The availability of the source code in libre software projects has been widely used, being the most frequent data source found in empirical research literature. The second position in number of research papers is occupied by the data offered by versioning systems, which has been mined primarily to obtain information about the maintenance and evolution of the software system. Next, we find bug-tracking systems as data source. In this regard, it should be noted that many works try to link data from this source and from CVS.

Mailing lists are under-represented if we consider their strategical importance in libre software projects. The reason for this is that besides the information that can be found in the headers of the e-mail messages, the content itself is difficult to analyze and to link with other artifacts. Future research should focus on solving these problems.

Software development web platforms such as SourceForge have been also widely used, specially for holistic studies that try to look at hundreds if not thousands of libre software projects. Information from all the other sources is completed by survey. Surveys give complementary information that cannot be obtained from the other data sources.

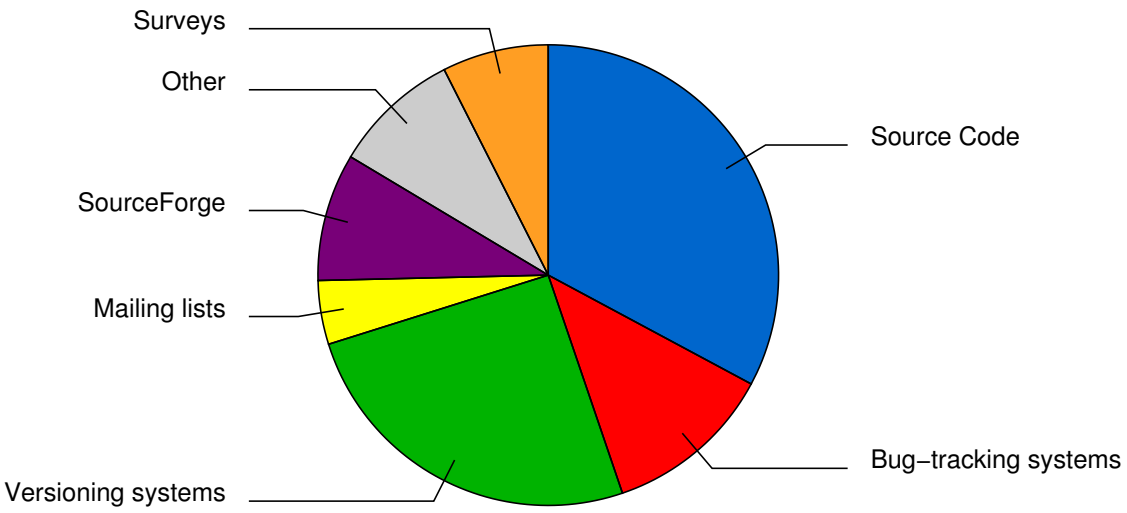


Figure 2.3: *Distribution of the data sources used in software engineering and related literature that has focused on libre software projects (Total: 67; works that include data from various sources have been counted twice).*

Chapter 3

Sources and data

If you torture the data enough, it will confess.
Ronald Coase

This chapter will describe in detail the data sources and the data obtained from them. As already noted in previous chapters, the fact that communication and organization is heavily tight to the use of telematic means makes the number of possible data sources grow beyond source code. Besides, the ability of having *memory* provides with the possibility of obtaining data from points in the past and to perform longitudinal analyses.

The concepts and findings that will be presented in the following sections are partly contributions of this thesis, although many of them relay heavily on the current state of the art. This is especially the case for the analysis of source code and versioning repositories as both have an ample literature. This chapter goes beyond that and is probably the first attempt to have a detailed description of the data sources that can generally be found for libre software projects on the Internet and the data that can be found in them.

3.1 Identification of data sources and retrieval

Before entering into detail into the description of the data sources, there are some previous steps that should be considered: identification and retrieval. It should be noted that there may be several ways of accessing the data, depending on the projects. This is because of the use of different tools and of having different usage conventions (for instance, different use of tags, comments, among others). The complexity and feasibility of both activities depends on the data source and on the project. Figure 3.1 gives a diagram that shows the steps that have to be accomplished for any source considered in our study.

Figure 3.1: *Whole process: from identification of the data sources to analysis of the data.*

In general terms, the identification of the data source depends mostly on its significance for the software development of a project. Hence, identifying the source code, the control versioning system, the mailing lists or the bug-tracking system is by no way problematic as it lies in the interest of the projects that feedback is provided by users in an easy and fast way. In these cases, the biggest drawback is the lack of historical data. Sometimes we only have a partial set of the data, and in the worst cases nothing at all. This situation is common for software releases, where finding historical versions of the software is sometimes not possible. Other situations where this might happen is when a development tool has not been used in the early stages of development. This is the case of many projects that start using a versioning system once the project has gained certain momentum. Having

only partial data can also be the result of a migration from one tool to another, losing in the way some information if not all. When researching libre software projects, these considerations have to be taken into account.

But there exist other data sources for libre software projects that are not so obvious and hence their identification is not that straightforward. For instance, organizational information that is embedded into some format and that is beyond the use of standard tools as versioning systems, mailing lists and bug tracking systems. In general, such type of information is project-dependent and can be only obtained for one project or a small number of them. This is the case for packaging systems such as the .deb format used in Debian and Debian-based distributions or the .rpm Red Hat package system in use in Red Hat and other distributions. But beyond this, we can find project-related information in other means such as the Debian Popularity Contest as we will see in section 3.6.2 or the Debian Developer database presented in section 3.6.3. Other data sources may also be considered; for instance, in KDE there is a file that is used to list all the ones who have write access to their versioning repository. Another example is given in literature by a study by Tuomi [Tuomi, 2004] in which the credits file of the Linux kernel are studied in detail. Identification of the data source requires in such cases specific knowledge on the project and is difficult if not impossible to be generalized.

In this work, we will mainly focus on data sources that can be obtained from the Debian project as it is a project sufficiently known to the author. Beyond Debian many projects offer other type of information that could be identified and would enrich the analysis.

Once the data source has been identified, it has to be retrieved to a local machine in order to be analyzed (see figure 3.1). Although this process may not seem to be very difficult at first, previous experiences have shown that some considerations and good practices should be followed in this step as reported by Howison et al. in the retrieval of information from the web pages hosted at SourceForge [Howison & Crowston, 2004].

In the next sections we will enter into detail in the process of data extraction and data storage once the data has been properly retrieved from the information source to a local machine.

3.2 Source Code

As software development projects, source code is the central point of all interactions, being a primary way of communication and playing a major signaling and coordination point. According to [Lanzara & Morner, 2003], source code "is transient knowledge: it reflects what has been programmed and developed up to that point, resuming past development and knowledge and pointing to future experiments and future knowledge.".

The study of the source code, as the main product of the software development process is a matter that has been done for over thirty years now. In the approach that has been followed in this work not only traditional source code (i.e. programmed in a programming language) has been taken into account, but also all the other elements that make the software, such as documentation, translation, user interface and other files.

Our analysis starts with a source code base that is stored in a directory (or alternatively in a compressed directory, usually in tar.gz or tar.bz2 format as it is common in the libre software world). After uncompressing the tarball, if needed, the hierarchical structure of the source code tree is identified and stored.

Then, files are grouped into several categories depending on its type (as will be described below) which allows for a more specific analysis depending on their type. This means, for instance that source code files in a programming language can be analyzed differently than images or documentation files. On the other hand, the discrimination for files with source code can be finer, identifying the programming language and offering the possibility of using alternative metrics depending on it. As a consequence, object oriented metrics could be applied to files containing Java code, but would not be required for files that are written in assembler language.

The whole process can be observed in figure 3.2: after (possibly) uncompressing, the directory and file hierarchy is obtained, then files are discriminated by their type and finally are analyzed, if possible taking into consideration the file type that has been identified in the previous step. In the following

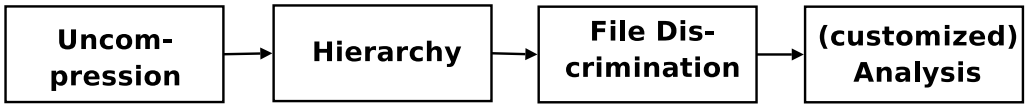


Figure 3.2: *Process of source code analysis.*

subsections the different steps are described more in detail.

3.2.1 Hierarchical structure

The structure of directories and files of a software program (and how it changes over time) has already been the focus of some research studies [Capiluppi, 2004; Capiluppi *et al.*, 2004b]. The idea is that the technical architecture and probably therefore the organization of the development team is mapped by the tree hierarchy of directories. So, from a directory hierarchy as shown in figure 3.3, we could infer the organizational structure of a libre software development project.

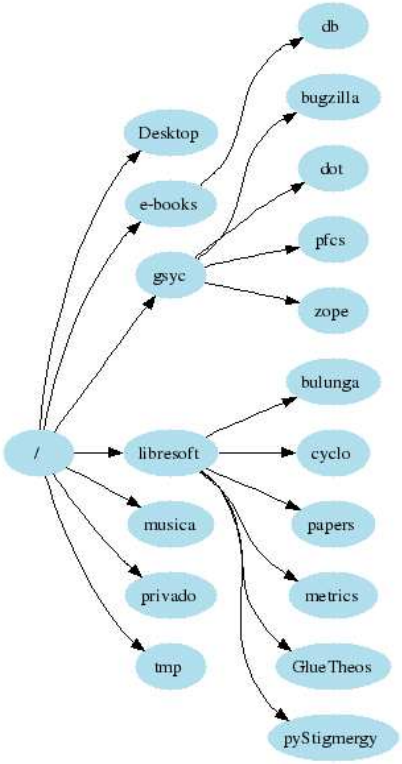


Figure 3.3: *Tree-like hierarchical structure.*

But beyond the structure of a directory tree, we are interested in aggregating data from a given point downwards all the paths that depend on it. This will allow having information on the different branches, identify if different teams work (as expected) on different parts of the tree, and finally perform other types of analyses.

Although this idea seems very promising, it raises the problem that relational databases do not naturally implement tree-like structures and queries may become too expensive and complex. A first attempt to implement a tree-like hierarchy in a database is to have a database table which contains all directories. For every entry, we will have the directory name, its unique id as primary key and the id of the parent directory as a foreign key.

If we require a query to obtain information on a branch, we will have to obtain first the id of the directory from where we are looking downwards. Then we will have to find all the directories that hang up from it by searching for those directories whose parent id is the one of the first directory.

This process should be performed recursively for all the directories that pend from the directories at this level until none of the directory ids can be found as parent ids (which means that there are no more pending directories). For a sufficiently large software project with a complex tree hierarchy this algorithm supposes a large amount of queries (one per directory in the branch). After studying this problem, we have found an efficient solution in the use of ad-hoc algorithms for trees and hierarchies as proposed by Celko [Celko, 2004].

3.2.2 File discrimination

File discrimination is a technique that is used to specifically analyze files on behalf of their content. The most common way of discriminating files is by using heuristics, which may vary in their accuracy as well as in the granularity of their results.

A first set of heuristics may determine the type of a file by considering its extension. File extensions are non-mandatory, but usually conventions exist so that the identification of the content of a file can be made easier and to enable the automatization of administrative tasks.

Hence, a first step for file discrimination consists of having a list of extensions that links to the content of the file. In this context, the .pl extension is indicative for a file that contains programming instructions while a .png can be considered as an image file. Of course, this can be done at several granularity levels, meaning that a .c file is a file that with high probability contains programming language, being that the programming language C code. Table 3.1 shows an excerpt of the list of file extensions that has been used for this thesis, while the complete reference can be consulted in appendix B.

We have selected a list of extensions and common file names (for which the column *Extension/file name matching* in table 3.1 is only an excerpt) which is matched against every file name. The file types we consider are documentation, images, translation (i18n), user interface (ui), multimedia and code files. For the latter type, we have considered a more detailed analysis and discriminate between source code that is part of the software application (code) from the one that helps in the building process (generally Makefiles, configure.in, among others) and from documentation files that are tightly bound to the development and building process (such as README, TODO or HACKING).

File type	Extension/file name matching
documentation	*.html *.txt *.ps *.tex *.sgml
images	*.png *.jpg *.jpeg *.bmp *.gif
i18n	*.po *.pot *.mo *.charset
ui	*.desktop *.ui *.xpm *.theme
multimedia	*.mp3 *.ogg *.wav. *.au *.mid
code	*.c *.h *.cc *.pl *.java *.s *.ada
build	configure.* makefile.* *.make
devel-doc	readme* changelog* todo* hacking*

Table 3.1: *(Incomplete) set of matches performed to identify the different file types. For an extended list, look at the appendix B*

A second step in the process of file discrimination includes inspection of the content of the files both to check if the identification made by means of matching file extensions is correct and to identify files that have no extension or whose extension is not included in the previous list.

In this case, heuristics are generally content-specific and may go more in depth depending on the detail of discrimination we are looking for. One of the most common ways to improve file discrimination by looking at the file content is to analyze the first line. There exist some convention in source code files that implicitly denote the programming language that they contain. For instance, in the case of a file written in the Python, *Bourne again* shell or Perl programming language, the first line could contain respectively the following information[1]:

[1]The location of the binaries may depend from system to system, although the standard location for them is the /usr/bin directory.

```
#!/usr/bin/python
#!/usr/bin/sh
#!/usr/bin/perl
```

In the case of programming languages, further information can be gained from the structure of the code, by the identification of specific keywords or other elements such as specific comments. For text files (especially the ones that are based on mark-up languages), tags and other specific elements may help in the identification process. Finally, other algorithms can be taken into account, as the information that returns the UNIX *file* command on the file type (which also identifies some of the binary formats, especially useful in the case of images).

Some of the previous discrimination techniques are already in use in some of the tools that have been presented in the chapter devoted to the state of the art, most notably in the SLOCCount tool (see section 2.2.3). As SLOCCount counts the number of lines of code it is only concerned with identifying source code files and identifying the programming language in which they are written in, not considering all other file types that we have taken into consideration in this work (documentation, translations, and other).

3.2.3 Analysis of source code files

The analysis of source code files is one of the most known in software engineering literature as we have already seen in the chapter 2. There exist an ample number of measures that can be and have been extracted directly from the source code, among other its length (in lines of code or source lines of code), complexity measures (as the popular ones proposed by Halstead [Halstead, 1977] and McCabe [McCabe, 1976]) or even composite metrics such as the Maintainability Index [Oman & Hagemeister, 1992].

In section 2.2.3 we have also seen that there exist some specific tools for the analysis of source code from the various points of view presented above. The availability of a certain range of tools for this purpose makes the conception of a tool that integrates all of them a primary task. The goals of the integration is to make it possible to extract all the metrics and facts from source code files by using several tools in a simple and most uniform way. The tools used to measure the code should be, if possible, used as *black boxes*, so that the integration tool does not need to know or adapt its inner functioning. In addition, the integration tool should handle the input to and the output from the measurement tools to ease its use.

That is precisely what we have addressed with GlueTheos[2], a tool designed and implemented for this thesis: to design a system with an architecture that allows the data retrieval and analysis of public software development data repositories. The structure of the GlueTheos tool is presented in figure 3.4, and consists of a module for downloading (if possible, with a periodical pattern) the sources to be analyzed, to examine the content of the sources on a file basis, to run the tools depending on the file type, to identify the results and store them properly in a relational database system and finally to provide results.

The current version can access CVS versioning repositories and archives of source packages (both in deb and rpm formats). It has been designed in a highly modularized way, so that adding new retrieval methods (from CVS or other data repositories) and analysis procedures is simple.

The file discrimination procedures that have been implemented are the ones that have already been presented in subsection 3.2.2. File discrimination allows to run the tools specifically on the files where this makes sense. Hence, if we had a tool that returns object oriented measures from Java files it would make no sense to run it on a shell script. This step optimizes then the analysis to be performed.

The next step is the heart of GlueTheos and consists of running the different tools on the source code and retrieving the data that these tools return. GlueTheos has been designed in a way in which

[2]GlueTheos is named after its purpose to glue different tools together in an easy way. Hence, this program is the *god*, *theos* in Greek, of gluing some already existing tools together.

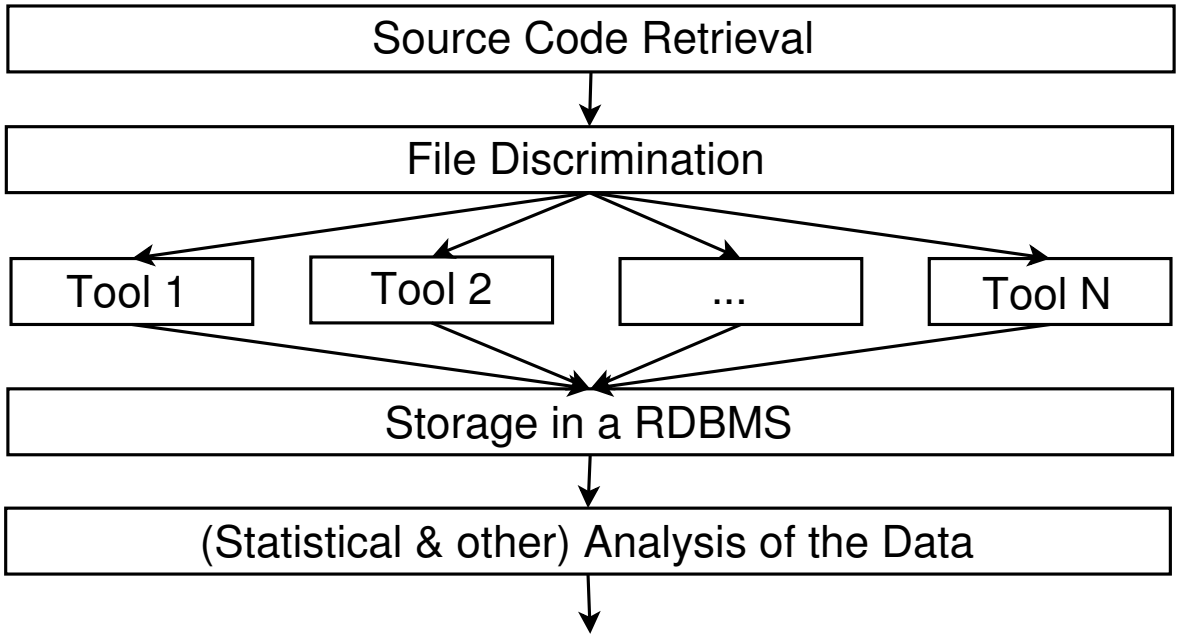


Figure 3.4: *Architecture of the GlueTheos tool.*

it does not require to adapt the tools it integrates, hence facing the complexity of the various ways of calling them and the various ways of obtaining their results. Both calling and returning has been solved following an object-oriented approach, so that for any tool only the differences have to be implemented.

The calling procedure requires information such as the way a tool has to be called (mainly the path to the executable), the input that the tool requires (usually a file or a directory) and the type of output that the tool returns (again, usually a file or a directory). Following excerpt of the configuration file shows an example of how the calling is configured for a tool that returns Halstead measures:

```
self.tool               = 'halstead'           # toolname
self.inputType          = 'file'               # 'file' or 'directory'
self.execution          = '/usr/bin/halstead'  # path for the tool
self.outputType         = 'file'               # 'file', 'directory' or 'both'
```

The returning methods depend on the *outputType* variable set above. If it is a file, the number of returning elements has to be given and the special character that is used to separate them (usually a tabulator, a white space or a comma). In general, the path that gives the filename of the file that has been analyzed is also returned, so its position (starting to count with 0) has to be specified. Finally, a list contains the field names that are used in the database, followed by another list that gives their type. This allows for easy conversion from the tool's output format to an SQL format suitable for the database. As an example, the configuration parameters for the Halstead measurement tool are given:

```
self.result_elements    = 5     # Number of results (including path)
self.path_order         = 0     # Position where the full path is (starting with 0)
self.elements_separator = '\t'  # Results are separated by
                                # (possibly: tabs, white spaces, commas, etc.)

self.elementList        =  ['file',
                            'program_length',
                            'program_volume',
                            'program_level',
                            'mental_discriminations']
```

```
self.elementListType    =  ['varchar(255)',  # file
                            'int(8)',        # program_length
                            'int(8)',        # program_volume
                            'int(8)',        # program_level
                            'int(8)']        # mental_discriminations
```

After retrieving and storing the data from external tools, GlueTheos has to consider only the data in the database to obtain statistical and other results from the data set. This includes some procedures to enhance the database structure in order to normalize the fields or to obtain intermediate tables with statistical information that is of common use.

3.2.4 Analysis of other files

Besides source code written in a programming language, we identify other artifacts that compose the sources of libre software projects. In section 4.5 we will see the many possibilities that arise from the study of those files, but other references to this issue may be found in related literature. Some authors have focused on the analysis of the change log files [Capiluppi *et al.*, 2003] as they usually follow a common pattern in libre software projects, although sometimes this pattern is slightly different from the standardized way used in GNU projects.

Translation files may be used to keep track of the amount of translation work that has been accomplished to the moment and hence have a quantitative manner of knowing the support of that software in a given language.

Regarding licenses, in addition of a reference to the licensing terms that can be found at the top of the code files, usually projects have a text file which includes the full text of the license. The filename may give enough evidence for the type of license that a project uses, but other ways can also be considered. One that we have been trying with is the use of a locality-sensitive hash like nilsimsa[3]. This type of hashes return codes with small changes for inputs that differ only slightly.

Finally, the amount of documentation for a software system could be a good topic for empirical research.

3.2.5 Authorship and dependency analysis

Usually, source code files contain copyright and license information in their first lines [Spinellis, 2003]. So, for instance, the notice in the apps/units.c file of the GIMP project shown below clearly states that the copyright holders are Spencer Kimball and Peter Mattis and that the license in use is the GNU General Public License:

```
/* The GIMP -- an image manipulation program
 * Copyright (C) 1995 Spencer Kimball and Peter Mattis
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
```

[3]The nilsimsa code can be retrieved from following URL: http://ixazon.dynip.com/%7ecmeclax/nilsimsa.html.

```
* Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.
*/
```

During the work on this thesis, we have been working with two software programs that extract authorship information from source code files. The first one is CODD, designed by Rishab A. Ghosh and implemented originally by Vipul Prakash (see section 3.2.5) and which we have been maintaining since the year 2001. CODD raises some limitations regarding authorship identification so we decided to create a new tool from scratch based on the heuristics given by CODD. This tools has been called pyTernity. In the following paragraphs we will describe both tools in detail.

CODD[4] is a tool that searches for authorship information in source code by tracking copyright notices and other information in the headings of files. It assigns the length (in bytes) of each file to the corresponding authors. CODD has been used in several studies since its first release in 1999 (Orbiten Survey [Ghosh & Prakash, 2000], source code survey done for the FLOSS study [Ghosh *et al.*, 2002b], among others). The process that CODD follows to obtain these results are shown in figure 3.5.

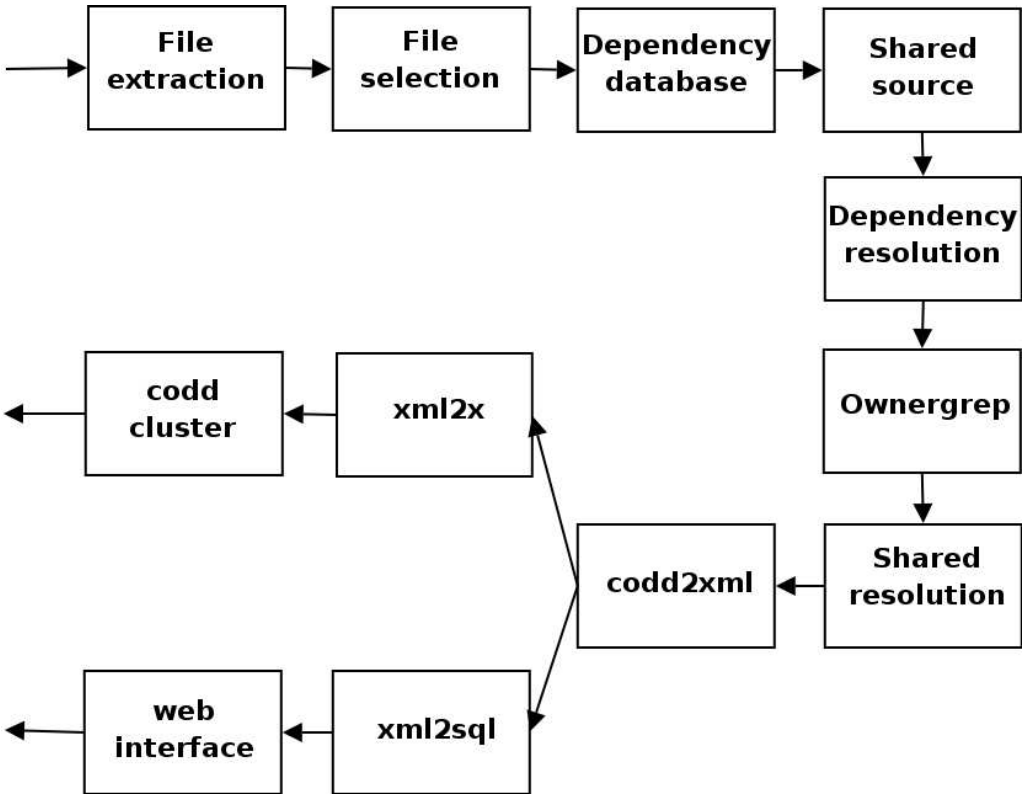


Figure 3.5: *Process of the CODD tool.*

File extraction is composed of the *init* subroutine which takes the source code package (or packages) that are given through the command line by the user, uncompresses them if necessary, and tries to identify recursively the files that the package contains.

During the file selection all source code files, documentation, interfaces and not-resolved implementations are taken together with their size in bytes, their MD5 sum and their relative route in the package and stored in the *codd*[5]. Files are selected by means of their extension, so for instance the .c file extension is categorized into source code files (usually they correspond to C files, while other extensions belong to documentation). CODD stores the *.h* files that have a *.c* in the same package as interfaces (the algorithm that is used here depends partially on the programming language that is being analyzed). Calls to an interface in source code files (for instance *.c* files for C) that do not have

[4]The most current version of CODD may be found at http://libresoft.urjc.es/index.php?menu=Tools&Tools=CODD.
[5]CODD uses as intermediate storage a file for each source package which are generally called the *codd* files.

their corresponding interface in the same package (a *.h* for C) will be classified in the non-resolved implementations category, that in a future step will be handled for dependency resolution.

In a third step two databases are created in order to find shared source code and dependencies. In the first one, named *codefile_signatures*, all the MD5 sums of the files are stored. The second one contains all the interfaces that were found in the previous step. MD5 is a type of hash that allows to know if two files are equal; if they are they will have the same MD5 hash value. MD5 hashes are very interesting when the source code file is exactly the same, but a single modification (i.e. when it is committed into the CVS of the new project the RCT-type id changes) makes it impossible to recognize it as a shared file. New hashes, as for instance the nilsimsa hash, are built in a way that they are linear with the changes so that a small change in a file results in a small change in its nilsimsa code.

In order to find shared source code, CODD runs another time through all codds and looks if the source code files appear more than once in the database (really it looks if their name and MD5 sum appears more than once). If this occurs, the file is located in at least two different packages. Following information will be added to the shared source code section in the codd (*shared*): (source code file, path, MD5, size) => package name.

CODD does a similar process as in the previous step to resolve dependencies. It will search for not-resolved implementations in the codds and compare their MD5 sums with the ones that are stored in the interfaces database. If there is a coincidence, the interface will be deleted from the non-resolved interfaces section in the codd and added to the resolved ones. Also, a list with all the packages where this interface is implemented will be inserted.

The owner grep block is the one that is responsible for looking for authorship contributions. It runs again through all source code and documentation files and scans authorship attribution by means of certain heuristics. Mainly the heuristics look for several patterns: email addresses [1], copyright notices [2] and software control versioning ids [3]. Information about the authors is stored in the *credits* section of the codds. The regular expressions that have been used are following:

```
[1] Email grep: [\d\w_\=\.\%]+?\@[\d\w\._\-]+?\.\w+?)(?=[\s:>\n\r\)]|$

[2] Copyright grep: .*copyright (?:\(c\))?[\d\,\-\s\:]+(?:by\s+)?([^\d]*)

[3] Id grep: (?:Id|Header).*?\d\d\:\d\d\:\d\d (\S+?) \S+?
```

Next, the resolution of shared source code is done. In the *shared* source code section of the codds we still have files and a list of packages that contain these files. As these files can only be assigned to a single package (in order to avoid double counting the contribution of an author), CODD looks for its author (running again the *ownergrep* algorithm) in the file and assigns it to the package in which the author is the main contributor.

The last blocks of figure 3.5 shows that the codds can be then transformed to an intermediate and independent format (as for instance XML) from which other analysis, correlation and clustering techniques may be done. There also exists a tool that does the transformation of the data from the codd format to SQL (including several normalization tasks for the resulting tables).

CODD is a very powerful tool, but it has some weaknesses. The most important one is that it lacks a way of merging the various ways in which an author may appear. So, authors may appear several times with different names or e-mail addresses. For instance, we have found that some developers have up to 15 e-mail addresses. In the case of companies, the same may happen; so, IBM or the MIT appear in several ways (up to ten times!) with slightly different wordings.

Cleaning of the data should also be enhanced. The heuristics that are used in CODD have proven to be very powerful, but cannot avoid that developers use different conventions to assign copyright. Most of these problems could be solved by a set of more powerful heuristics.

Both limitations were so important, that we decided to create a new tool from scratch that focused on these issues and which has been named pyTernity[6]. The architecture of pyTernity is identical to

[6]The most current version of pyTernity may be found at http://libresoft.urjc.es/index.php?menu=Tools&Tools=pyTernity.

the one described for CODD as it can be seen from figure 3.6, although it lacks of all the procedures for identifying dependencies among files.

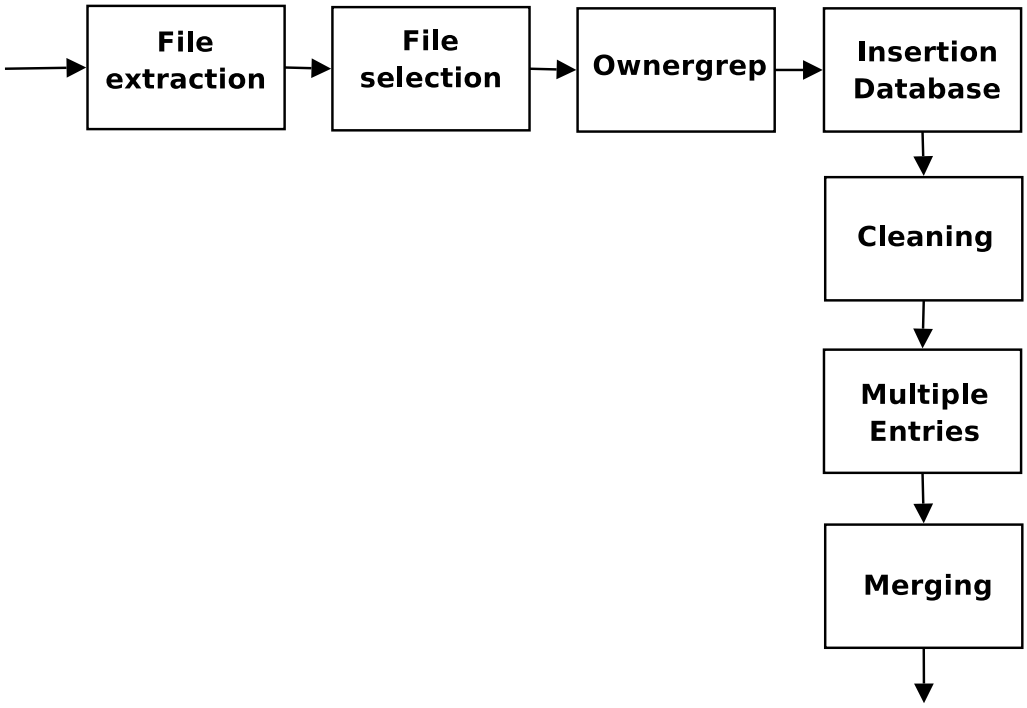


Figure 3.6: *Process of the pyTernity tool.*

The most innovative elements are the ones that consider data cleaning and the identification of multiple entries. For the former, entries in database are removed from elements that make them be different; this goes from additional white-spaces to the avoidance of dots. Some heuristics have been set up for this, although they have been complemented with a database of frequent changes. Cleaning includes splitting up an entry when it is due to two or more authors. So, the entry "Spencer Kimball and Peter Mattis" will result in two, one for Spencer Kimball and another one for Peter Mattis. If this is the case, both names appear as authors of the file and get attributed half of its length (in bytes or lines of code).

The latter part comprises the identification of multiple entries. Developers may appear in several ways, making results very unsatisfactory. The first efforts in this sense went into the construction of a large database where the various entries identified for a given developer were merged into a unique one. This has proven to enhance results in a prominent way, but the inclusion of new methods and the rising complexity has finally made us decide to have an external tool, which will be described in section 3.7.2, and that returns a unique identifier for any given identity. It is responsibility of this external tool to track all developers and to manage them properly.

Once cleaning has been performed and multiple entries have been identified, pyTernity merges the entries in the database so that authors appear only once in a file. This procedure implies to add all the contributions by an author, so it adds the lengths of each entry (in bytes or lines of code).

3.2.6 Dependency analysis

There are source code files that include header files in order to make use of the functions they provide. Header files declare functions that can be used by other files, but they may not contain their implementation/definition, which can be located in another source code file. Summarizing, we can see that we can have up to three different agents that have different relations to the functions: *uses* (final source code file), *declares* (header file) and *defines* (another source code file).

Codd-dependency is a tool, designed by Rishab A. Ghosh and by the author of this thesis, that has been implemented in order to make the analysis of this kind of software dependencies possible. It

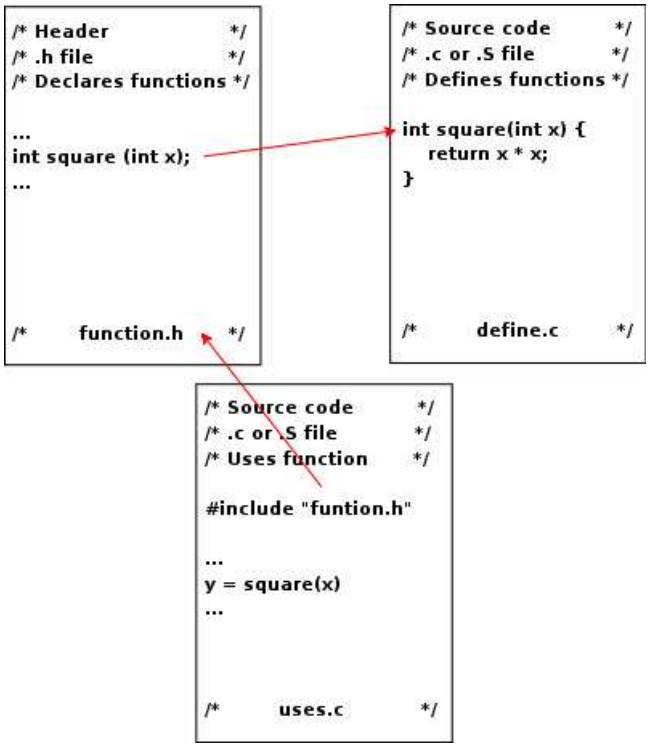


Figure 3.7: *Code dependency.*

consists of several modules, being codd-dependency the one with which the user interacts with. The main structure of codd-dependency is as shown in figure 3.8.

Hence, codd-dependency.py is a global process script which just calls all other submodules and stores their results into a database. The user can interact with it using command-line arguments or by setting several configuration parameters directly in the code.

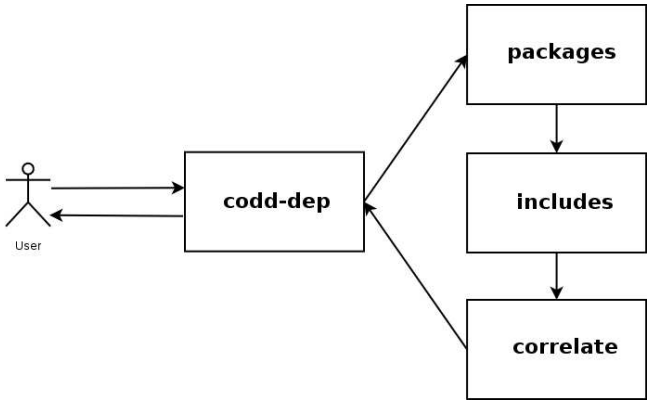


Figure 3.8: *Code dependency.*

packages.py prints information of the files in SQL format for all tarballs stored in a directory given by the command line input. Basically it *untars* the tarballs that exist in a given directory, and looks for C-code files by looking at the file extensions (which usually are .c, .h and .S). These files are given an unique id and stored into a database together with their filename (which includes the whole inner path in the package) and the name of the package they belong to.

includes.py extracts includes, function definitions and function declarations out of C-coded files and C headers. The algorithm is very simple: there are some predefined patterns that will be looked for in the files. Files that include headers are targeted by [1], while function definitions are done by [2] and [3]. Notice that [3] is the same as [2], but without an ending '{' as depending on the project

coding standards, the brackets can be on the same line as the function definition or in the next one.

If the file is a header file (.h), it also looks for function declarations [4]. The regular expressions that have been used are given below:

```
[1] Include pattern: '^#[\t ]*include[\t ]+[<"]([a-zA-Z0-9_/\.]+)'

[2] Function definition pattern 1: '[A-Za-z_\* ]+ [\*]?([A-Za-z0-9\-_]+)\(.*\) {'

[3] Function definition pattern 2: '[A-Za-z_\* ]+ [\*]?([A-Za-z0-9\-_]+)\(.*\)'

[4] Function declaration pattern: '^[A-Za-z_\* ]+ [\*]?([A-Za-z0-9\-_]+)\(.*\);'
```

There is some logic in order to avoid to identify structs, unions, registers and other elements as definitions or declarations. Finally it gets the file id from the files table (created before with the packages.py script) and inserts file-include, file-definition and file-declaration pairs into the database for future matching. At the end of this step we should have in the database a list of files and the headers they include, a list of files and the functions they define and a list of (header) files and the functions they declare.

correlate.py does the final step of matching headers and (source code) files where the functions used in other (source code) files have been defined. Basically the algorithm is as follows:

The program gets from the database all headers included in (source code) files [1]. For each obtained header it looks in the database for the filename and defined function(s) of all functions declared in the header and defined in other (source code) files [2]. Finally, for every match it prints out the filename of the header (declaration) and the filename of the defining file (definition) as well as the number of times that these two files have matched.

3.3 Versioning system meta-data

Versioning systems are used to manage file versions in a software development project. Thus they allow to track changes and past states of a software project. So, obtaining the current and any past state of the code is made possible by the use of a versioning system. This allows to make source code analyses as we have presented them in the previous section in a longitudinal manner and to extract facts on the evolution of a software project.

But beyond this, versioning systems store a set of meta-data of the changes. These meta-data can be tracked and analyzed. This information is usually related to the interactions that occur among developers and the versioning systems. In general the information is only related to actions that comprehend write access while reading (downloading the sources) or obtaining other information (diffs, among others) cannot be tracked in that way.

Together with this thesis, a software tool has been built that analyzes the interactions that occur between developers and the most used versioning system used in libre software projects at the current time, the Concurrent Versioning System CVS[7]. Fortunately, there are a lot of libre and non-libre software projects in this situation: for instance, more than 10,000 projects hosted at SourceForge used it [Healy & Schussman, 2003], and the 11 largest projects in Debian 2.2 [González-Barahona *et al.*, 2001] use versioning systems (all of them CVS except for Linux that uses Bitkeeper, a proprietary versioning solution[8]). This tool, which has been labeled CVSAnalY, is based on the analysis of the CVS log entries and implements all the theoretical details that will be presented in this section.

In CVSAnalY any interaction -also called commit- a commiter[9] does with the central versioning system repository is logged with following data associated: commiter name, date, file, revision number,

[7] The next-generation versioning system that is becoming widely used in the libre software world is Subversion (http://subversion.tigris.org). At the time of writing these lines, our tool offers only partial support for Subversion repositories.

[8] As of June 2005, the Linux project moved to another versioning tool developed by the Linux developers.

[9] A commiter is a person who has write access to the repository and does a commit -an interaction- with it at a given time.

lines added, lines removed and an explanatory comment introduced by the commiter. There is some file-specific information that can also be extracted, as for instance if the file has been removed[10]. On the other hand, the human-inserted comment can also be parsed in order to see if the commit corresponds to an external contribution or even to an automated script.

Basically CVSAnalY consists of three main steps, preprocessing, insertion into database and post-processing, but they can be subdivided into several more as it has been done in figure 3.9.

Figure 3.9: *Process of the CVSAnalY tool.*

3.3.1 Preprocessing: retrieval and parsing

Preprocessing includes downloading the sources from the CVS repository of the project in study. Afterwards, aggregated modules[11] have to be removed to avoid counting commits several times. Once this is done, the logs are retrieved and parsed to transform the information contained in log format into a more structured format (SQL for databases or XML for data exchange).

Besides the information for every commit, other data obtained from the parsing requires some attention. Although commiters seldom change their username, sometimes this happens, so the various usernames have to be merged into a unique one. For instance, in the KDE project commiters usually get a CVS account prior to a *kde.org* e-mail address. If a developer is afterwards assigned an e-mail address the username of e-mail and CVS have to be identical for organizational and practical reasons. If the username in the e-mail address is different from the CVS username, CVSAnalY syncs with the former one and the actions done with both usernames are considered as done by a unique developer.

The following is a CVS log excerpt for the AUTHORS file of the KDevelop project[12]. It gives the last 6 revisions (from revision 1.45 to 1.49) done during the last months of the year 2003 until mid-2004.

```
[...]

RCS file: /mirrors/kde//kdevelop/AUTHORS,v
Working file: /mirrors/kde//kdevelop/AUTHORS
head: 1.49
branch:
locks: strict
access list:
keyword substitution: kv
total revisions: 103;   selected revisions: 103
description:
----------------------------
revision 1.49
date: 2004/06/21 18:57:13;  author: rgruber;  state: Exp;  lines: +4 -0
Added self
----------------------------
revision 1.48
date: 2004/02/24 14:42:59;  author: dagerbo;  state: Exp;  lines: +5 -1
```

[10] In a versioning system there is actually no file deletion. In the case of CVS, files that are not required anymore are stored in the Attic and may be called back anytime in future.

[11] Aggregated modules are modules that are shared between other modules. Such modules generally include system-wide administration and scripts. This information is kept in the CVSROOT/modules file.

[12] KDevelop is an IDE (Integrated Development Environment) for KDE. More information can be obtained from http://kdevelop.org/.

```
Added self :)
----------------------------
revision 1.47
date: 2004/02/15 22:40:33;  author: aclu;  state: Exp;  lines: +3 -3
Some more credits update.
----------------------------
revision 1.46
date: 2004/02/15 22:02:33;  author: geiseri;  state: Exp;  lines: +6 -0
I guess I need to accept the blame for these things...
----------------------------
revision 1.45
date: 2003/11/01 11:47:30;  author: dhaumann;  state: Exp;  lines: +4 -1
branches:  1.45.2;
KTabWidget -> KDevTabWidget,
added author.

[...]
```

While being parsed each file is also matched for its type. Usually this is done by looking at its extension, although other common filenames (for instance README or TODO) are also looked for. The goal of this separation is to identify different contributor groups that work on the software, so besides source code files the following file types are also considered: documentation (including web pages), images, translation (generally internationalization and localization), user interface and sound files. Files that don't match any extension or particular filename are accounted as unknown. This discrimination follows the criteria that have been presented in section 3.2.2, although it lacks the possibility of looking at the content of the files as we only consider filenames (because this is the only information that appears in the CVS logs).

CVS also has some peculiarities when introducing contents for the first time (this action is called initial check-in). The initial version (with version number 1.1.1.1) is not considered in our computation as it is the same as the second one (which has version number 1.1). The number of aggregated and removed lines in CVS are computed from this initial version. This means that the first commit (the initial check-in) logs as if 0 lines were added. This does not correspond to reality. In order to obtain the actual number of LOCs in the first version we count the LOCs by means of the UNIX wc tool of the latest version, subtracting the added lines and adding the removed lines of all the other commits.

Comments attached to commits are usually forwarded to a mailing list so that developers keep track of the latest changes in CVS. Some projects have established some conventions so that certain commits do not produce a message to the mailing list. This happens when those commits are supposed to not require any notification to the rest of the development team. A good example of the pertinent use of *silent* commits comes from the existence of bots that do several tasks automatically.

In any case, such conventions are not limited to non-human bots, as human commiters usually may also use them. In a large community -as it is the case for the ones we are researching- we can argue that *silent* commits can be considered as not contributory (i.e. changes to the head of the files, for instance a change in the license or the year in the copyright notice, or moving many files from one location to another). Therefore, we have set a flag for such commits in order to compute them separately or leave them out completely in our analysis.

For instance, the developers of the KDE project mark such commits with the comment *CVS_SILENT* as it can be seen from following log excerpt extracted from the kdevelop_scripting.desktop file of the KDevelop CVS module. In this case it is due to a change to a *desktop* file, a file type that is related to the user interface. Being this change not considered interesting for other developers to know about, the author of this commit decided to make this commit *silently*.

```
[...]
```

```
RCS file: /mirrors/kde//kdevelop/kdevelop_scripting.desktop,v
Working file: /mirrors/kde//kdevelop/kdevelop_scripting.desktop
head: 1.24
branch:
locks: strict
access list:
keyword substitution: kv
total revisions: 30;    selected revisions: 30
description:
----------------------------
revision 1.24
date: 2005/03/28 03:29:25;  author: scripty;  state: Exp;  lines: +2 -2
CVS_SILENT made messages (.desktop file)
----------------------------
revision 1.23
date: 2005/03/27 04:05:51;  author: scripty;  state: Exp;  lines: +4 -0
CVS_SILENT made messages (.desktop file)
----------------------------

[...]
```

Write access to the versioning system is not given to anyone. Usually this privilege is granted only to those contributors who have reached a compromise with the project and the project's goals. But external contributions -commonly called patches, that may contain bug fixes as well as implementation of new functionality- from people outside the ones who have write access (commiters) are always welcome.

It is a widely accepted practice to mark an external contribution with an authorship attribution when committing it. Thus, we have constructed certain heuristics to find and mark commits due to such contributions. The heuristics we have set up are based on the appearance of two circumstances: patch (or patches in its plural form) together with a preposition (from, by, of, and other) or an e-mail address or an indication that the code had been attached to a bug fix in the bug-tracking system. The regular expressions that have been used are following:

```
[1] patch(es)?\s?.* from
[2] patch(es)?\s?.* by
[3] patch(es)?\s.*@
[4] @.* patch(es)?
[5] 's.* patch(es)?
[6] s' .* patch(es)?
[7] patch(es)? of
[8] <.* [Aa][Tt] .*>
[9] attached to #
```

As an example, the following slightly modified excerpt taken from the kdevelop.m4.in file from the KDevelop module in the KDE CVS repository shows a patch applied by a commiter with username dymo that was submitted originally by Willem Boschman:

```
[...]
----------------------------
revision 1.39
date: 2004/06/11 17:07:57;  author: dymo;  state: Exp;  lines: +3 -3
Applied patch from Willem Boschman -
fix builddir != srcdir configuration problem.
```

```
---------------------------
[...]
```

All these efforts have in common that they perform text-based analysis of the comments attached by commiters to the changes they perform. The range of possibilities in this sense is very ample. For instance, Mockus et al. tried to identify the reasons for changes (classifying changes as adaptative, perfective or corrective) in the software using text-analysis techniques [Mockus & Votta, 2000].

3.3.2 Data treatment and storage

Once the logs have been parsed and transformed into a more structured format, some summarizing and database optimization information is computed and data is stored into a database.

Usually the output of the previous parsing consists of a single database table with an entry per commit. This means that information is stored in a raw form, containing the table possibly millions of entries depending on the size and age of a project[13]. Information is hence in a raw format and in an inconvenient way if we consider getting statistical information for developers and projects from it.

A first step in this direction is to make use of normalization techniques for the data. In this sense, commiters are assigned a unique numerical identification and if further granularity is needed, procedures have been implemented to do the same at the directory and file level. For the sake of optimization this has been introduced during the parsing phase so additional queries have not to be performed. The next step is to gather statistical information on both commiters and modules. These additional tables will give detail on the interactions by contributors or to modules, which is one of the most frequent information that is asked.

Additional information that makes longitudinal analyses possible is the evolution of contributions by developers and to modules. Hence, the same statistical queries that have been obtained for commiters and modules for the summarizing tables can be obtained in a monthly or weekly basis since the date the repository was set up.

On the other hand, unfortunately CVS does not keep track of which files have been committed at the same time. The absence of this concept in CVS may bring some distortion into our analysis. We have therefore implemented the sliding window algorithm proposed by German [Germán, 2004d] and Zimmermann et al. [Zimmermann & Weissgerber, 2004] that identifies atomic commits (also known as *modification requests* or transactions) by grouping commits from the CVS logs that have been done (almost) simultaneously. This algorithm considers that commits performed by the same commiter in a given time interval (usually in the range of seconds to minutes) can be considered as an atomic commit (see figure 3.10 for more details). If the time window is fixed, the amount of time that is considered from the first commit to the last one is a constant value. For a sliding time window, the time interval is not constant; the time window is restarted for every new commit that belongs to the same transaction until no new commit occurs in the (new) time slot [Zimmermann & Weissgerber, 2004].

The post-process is composed of several scripts that interact with the database, analyze statistically its information, compute several inequality and concentration indexes and generate graphs for the evolution over time for a couple of interesting parameters (commits, commiters, LOCs...). Results are shown through a publicly accessible web interface that allows easy inspection of the whole repository (general results), a single module or by commiters. Therefore results are available for remote analysis and interpretation by project participants and other stakeholders.

The output of this stage will be present in several places in the next chapter devoted to the analysis of libre software projects. Figure 3.11 gives a view of the general statistics as they can be accessed through the web interface of CVSAnalY for the KDE project[14].

[13] For instance, as of April 2005, we have computed over 7 million commits to the CVS of the KDE project. The number of commits for other projects such as GNOME or Apache is also in the order of magnitude of millions of commits.

[14] http://libresoft.urjc.es/cvsanal/kde3-cvs.

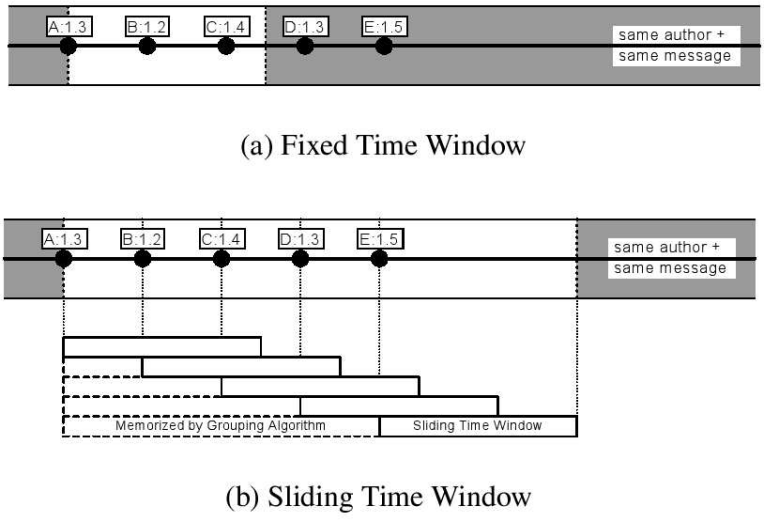


Figure 3.10: *Fixed vs. sliding time window algorithm for the identification of atomic commits. Source: [Zimmermann & Weissgerber, 2004].*

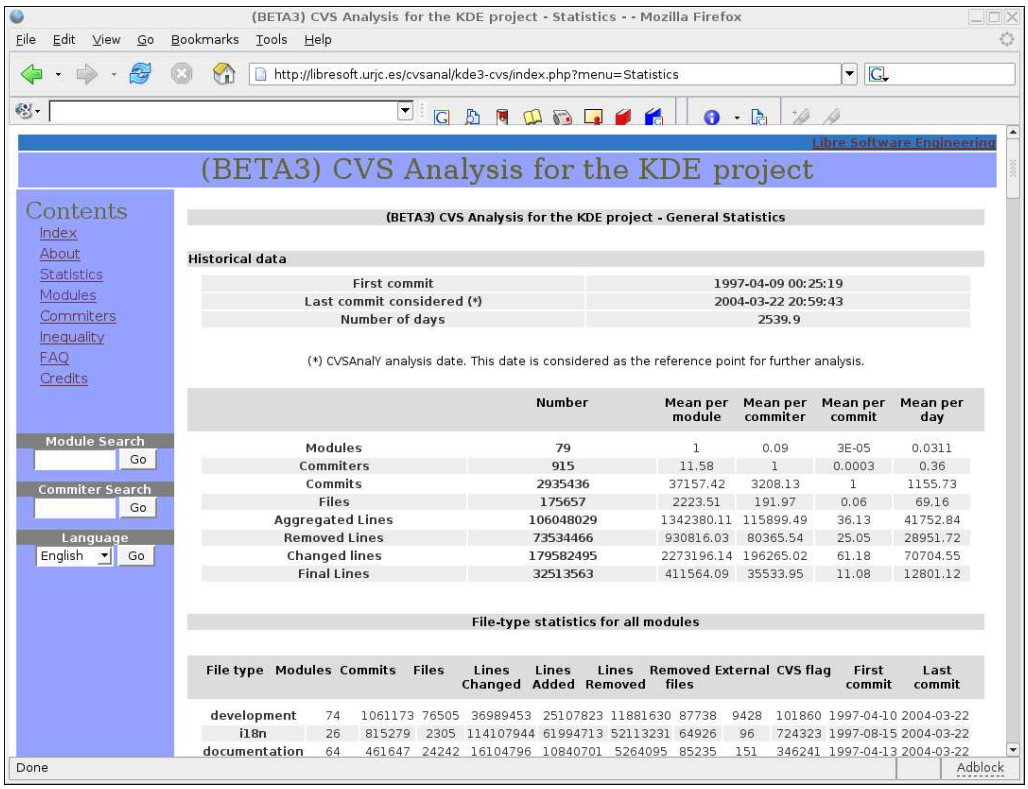


Figure 3.11: *Screenshot of the CVSAnalY web interface for the KDE project. Data is from April 2004.*

3.3.3 Software Archaeology

Based on the meta-data that versioning systems offer, we are able to follow on a per-line basis who and when a line was modified last. So, for instance, CVS has a command line option which shows the revision where each line was modified last, giving the date and commiters responsible of that change. This option is labeled *annotate* and is available for other versioning systems as the next-generation Subversion (where it is called *blame*[15]). Next, a (slightly modified) excerpt of the annotated output for the src/keymap.c file from the GNU Emacs project can be found:

```
[...]
1.246    (pj      13-Nov-01): Optional arg STRING supplies menu name for the keymap
1.246    (pj      13-Nov-01): in case you use it as a menu with `x-popup-menu'. */)
1.246    (pj      13-Nov-01):      (string)
1.8      (rms     11-Sep-92):      Lisp_Object string;
1.8      (rms     11-Sep-92): {
1.8      (rms     11-Sep-92):   Lisp_Object tail;
1.8      (rms     11-Sep-92):   if (!NILP (string))
1.8      (rms     11-Sep-92):     tail = Fcons (string, Qnil);
1.8      (rms     11-Sep-92):   else
1.8      (rms     11-Sep-92):     tail = Qnil;
1.1      (jimb    06-May-91):   return Fcons (Qkeymap,
1.137    (rms     13-May-97):     Fcons (Fmake_char_table (Qkeymap, Qnil), tail));
1.1      (jimb    06-May-91): }
[...]
```

The structure of the CVS annotate output is as follows: the first column contains the file revision corresponding to the line, then in brackets first we have the username and afterwards the date of the commit and finally the content of the line.

We have introduced some error-correction routines that check for common errors found when mining data from CVS. So for instance some lines have a date that does not correspond to reality, probably due to a temporary misconfiguration of the server clock.

Once the data has been parsed and cleaned of comments, blank lines and errors, we normalize the data and insert it into a database server which we query for statistical information on the data set. This is performed by a set of scripts that are responsible for the generation of the graphs that will be shown in section 4.4 with some selected case studies.

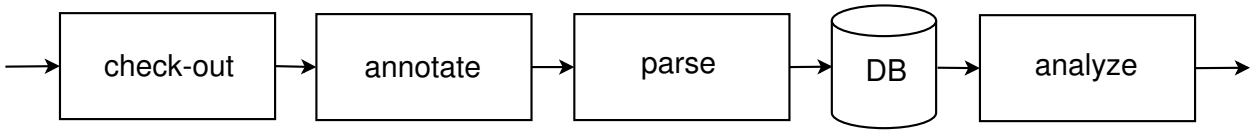


Figure 3.12: *Process of the DrJones, a tool conceived to analyze projects from a software archaeology point of view.*

As the whole process, from source retrieval to graph and index generation, can be made automatically and in a non-intrusive way we have put all the scripts together and have released a software named DrJones[16]. Figure 3.12 gives a schematic representation of the steps that DrJones goes through. DrJones is released under a libre software license, so that independent research groups have the possibility to use, and enhance it, comparing the results.

The approach we have presented has the limitation of being only applicable to software projects that provide over a versioning system. If the project to analyze does not make use of one, but we

[15] *annotate*, *ann* and *praise* are alternative names for the *blame* Subversion command.

[16] More information about the DrJones tool as well as the software can be found at: http://libresoft.urjc.es/index.php?menu=Tools&Tools=DrJones.

have the historic versions of it (and the dates of their release), there exists the possibility of building a versioning system from them. This process can be automatized, although some information from the original analysis will be missing as for instance the real date of the modification for lines (which will be set to the one where the release took place) and information regarding authorship.

Other problems arise because of certain characteristics (or lack of them) in the versioning system. For instance, in CVS there exists no way for the CVS client to move files from one directory to another, so users have to delete and create a file again in the new location which actually makes our software archaeology approach error-prone. Newer versioning systems, such as Subversion, do not have this limitation.

Finally, a third limitation is the insertion of external code. Vertical lines will appear when a large amount of lines of code will be added at one time (as we will see for some of the case studies in section 4.4). This inserts errors in the analysis regarding the date and authorship. On the other hand, although the starting date of this code is not accurate it is code that has to be maintained, so this limitation should not be that problematic for our purposes.

3.4 Mailing lists archives (and forums)

Mailing lists and forums are the main key elements for information dissemination and project organization in libre software projects. Without almost any exception, libre software projects provide over one or more mailing lists. Depending on the project, many mailing lists may exist for several target audiences. So, for instance, SourceForge recommends to open three mailing lists: a technical one for developers, another one to give support to users and and a third one that is used for announcing new releases.

Mailing lists are programs that forward e-mail messages they receive to a list of subscribed e-mail addresses. More sophisticated mailing list managers have plenty of functionality which allows for easy subscription, desubscription, storage of the messages that have been sent (known as the archives), and avoidance of spam, among others.

Forums are web-based programs that allow visitors to interact in a similar manner as in an e-mail thread with the difference that in this case all the process goes through HTML forms and that results are visible on the web.

Both, mailing lists and forums are based on similar concepts and their differences lie in their implementation and the need of different clients for participating in them. Mailing lists require the use of an e-mail client, while forums can be accessed through web navigators. As their concept is the same, there exist some software programs that transform mailing lists messages to a forum-like interface and vice-versa. Because of that, in this work we will only focus on mailing lists, specifically on one of the most used mailing lists managers called GNU Mailman[17] and the RFC 822 (also known as MBOX) format in which it generally stores and publishes the archives.

3.4.1 The RFC 822 standard format

As mentioned above, generally all mailing list managers offer the possibility of storing all posts (the archives) and making them publicly available through a web interfaces. This offers the possibility for newcomers to go through the history and to gain knowledge on technical as well as organizational details of a project.

The archives are also offered in text files following the MBOX format. MBOX is a format used traditionally in UNIX environments for the local storage of e-mail messages. It is a plain text file that contains an arbitrary number of messages. Each message is composed of a special line followed by an e-mail message in the RFC-822 standard format. The special line that allows to differentiate messages consists of the keyword “From” followed by a blank space, the poster’s e-mail address, another blank space and finally the date the message was sent. The RFC-822 format can be divided into two parts: (a) headers, that contain information for the delivery of the message and (b) the content, which is the

[17] The MailMan’s project web site can be found at following URL: http://www.gnu.org/software/mailman/.

information to be delivered to the receiver; the standard only allows lines of text, so filtering has to be implemented if an image or other information is attached.

Below is an excerpt of a post sent to a mailing list that has been stored following the RFC-822 standard. Is is an automatic message sent April 30 2005 to the GNOME CVS mailing list. This list keeps track of all the commits that are done to the CVS versioning system of the GNOME project. This assures that subscribers are aware of the latest changes in the CVS. The content of the message, the description of the modification that had been performed, has been omitted in the excerpt.

```
From gnomecvs@container.gnome.org  Sat Apr 30 20:16:38 2005
Return-Path: <gnomecvs@container.gnome.org>
X-Original-To: cvs-commits-list@mail.gnome.org
Delivered-To: cvs-commits-list@mail.gnome.org
To: cvs-commits-list@mail.gnome.org
X-CVS-Module: marlin
Message-Id: <20050501001636.0C5EA165E4A@container.gnome.org>
Date: Sat, 30 Apr 2005 20:16:36 -0400 (EDT)
From: gnomecvs@container.gnome.org (Gnome CVS User)
X-Virus-Scanned: by amavisd-new at gnome.org
Cc:
Subject: GNOME CVS: marlin iain
X-BeenThere: cvs-commits-list@gnome.org
X-Mailman-Version: 2.1.5
Precedence: list
Reply-To: gnome-hackers@gnome.org
List-Id: CVS Logs <cvs-commits-list.gnome.org>
List-Unsubscribe: <http://mail.gnome.org/mailman/listinfo/cvs-commits-list>,
 <mailto:cvs-commits-list-request@gnome.org?subject=unsubscribe>
List-Archive: </archives>
List-Post: <mailto:cvs-commits-list@gnome.org>
List-Help: <mailto:cvs-commits-list-request@gnome.org?subject=help>
List-Subscribe: <http://mail.gnome.org/mailman/listinfo/cvs-commits-list>,
 <mailto:cvs-commits-list-request@gnome.org?subject=subscribe>
X-List-Received-Date: Sun, 01 May 2005 00:16:38 -0000

[Here comes the body of the post which has been omitted in this excerpt]
```

From the message excerpt above, we can see some of the headers that are described in the standard. The most important ones are following:

- From: gives the e-mail address of the sender. Sometimes it also provides the real name of the person that submitted the message. Depending on the configuration of the e-mail client, the real name may appear in brackets just after the e-mail address. But the most common configuration is given by the real name followed by the e-mail address as described next: *real name* $<$ *username@example.com* $>$.
- Sender: Specifies the address of the responsible entity for the last transmission. If it is the same as the one in the "From:" header, it should be omitted.
- Reply-To: Address to which the author of the message wants to be replied.
- To: E-mail address(es) of the receiver(s) of this message. If more than one receiver appear, they have to be separated by commas.
- Cc: E-mail address(es) of the receiver(s) that should receive a copy of the message. Again, if more than one receiver is supposed to get a copy, a comma-separated list is given.

- Bcc: Addressee(s) with *carbon* copy. Other receivers of the message won't know that the e-mail address(es) listed here will receive a copy of this message. Of course, this field will not appear in the MBOX of the receiver, so we will not have this is information in our analysis.
- Subject: Subject of the message. Usually contains a brief description of the topic.
- Received: Contains the address of the intermediate machine that has transfered the message. The date of the action is also given. This field can give valuable information about the server clock.
- Date: Gives when the message was sent. The clock that is used is the sender machine, so if it misconfigured it may be wrong. Attached to the date, sometimes the timezone is given. This gives information about the local clock.
- Message-ID: Unique identifier of this message. The uniqueness is guaranteed by the emitting host.
- In-reply-to: Identifier of the parent message to which the current one is a response. This field allows to identify conversation threads.
- References: Stores identifications (message-IDs) of all the other messages that are part of the conversation thread.

In addition to the data that can be found in the headers, some other information could be obtained from analyzing the content of the messages. This idea is similar to the one presented in subsection 3.3.1 for the comments attached to CVS commits.

3.5 Bug-Tracking systems

Bug-tracking systems are used in libre software projects to manage the incoming error and feature enhancement reports from users and co-developers. In opposition to versioning systems and mailing lists, the use of bug-tracking systems is relatively new and the most known tool in this area is BugZilla, a bug-tracking system developed by the Mozilla project and that has been adopted by other large projects as well. Hence BugZilla is the system we study in this work, although conceptually all other systems should work similarly.

BugZilla allows to manage all bug reports and feature requests by means of a publicly available web interface. Besides the reports, it also offers the possibility of adding comments so that developers may ask for further information about the error or other end-users may comment it. Beyond BugZilla, other tools exist with similar features, as for instance GNATS (the one used in the FreeBSD project). SourceForge and other web platforms that support software development have implemented their own bug-tracking systems for the projects they host.

3.5.1 Data description

BugZilla stores in its database specific information for each bug report. The fields that can be usually found are following[18]:

- Bugid: Unique identifier for any bug report.
- Description: Textual description of the error report.
- Opened: Date the report was sent.
- Status: Status of the report. It can take one of the following status: new, assigned (to a developer to fix it), reopened (when it has been wrongly labeled as resolved), needinfo (developers require more information), verified, closed, resolved and unconfirmed.

[18]The ones shown next are the ones that can be found for the GNOME BugZilla system. BugZilla can be adapted and modified, so the fields may (and will) change from project to project.

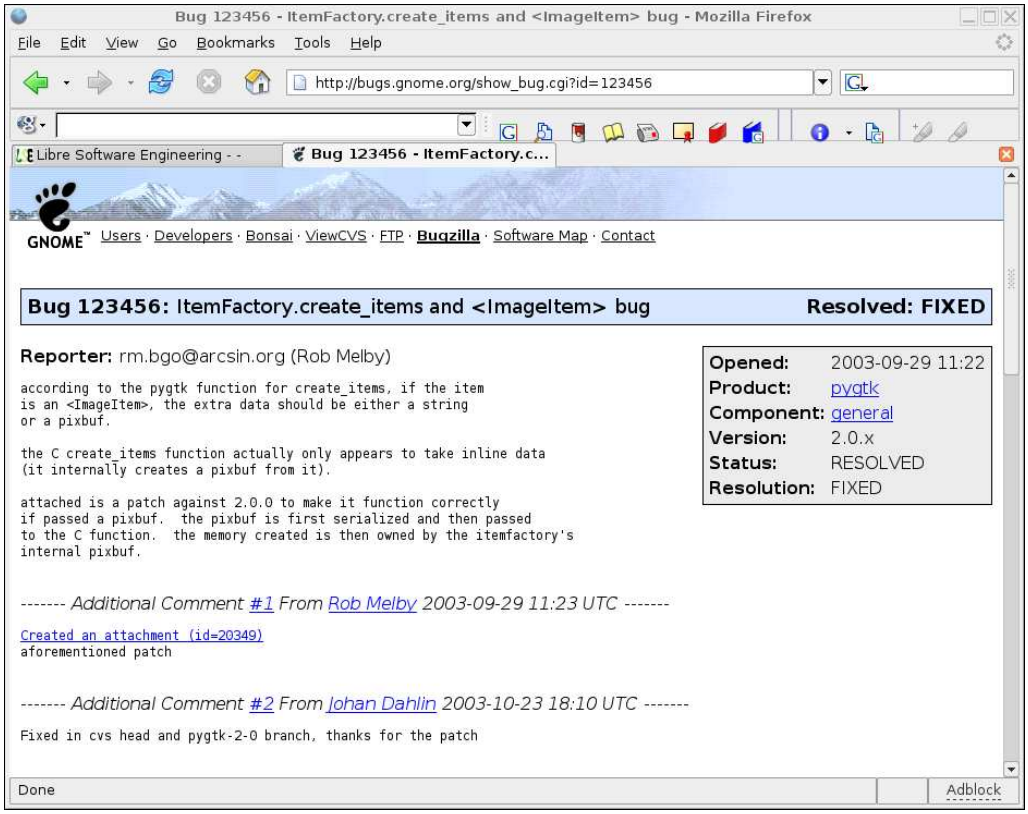


Figure 3.13: *Screenshot of GNOME's BugZilla web interface. The bug shown is bug #123456 (July 2005).*

- Resolution: Action to be performed on the bug. It can take following status: obsolete (will not be fixed as it is a bug to a previous, already solved issue), invalid (not a valid bug), incomplete (the bug has not been completely fixed), notgnome (the bug is not of GNOME, but of a component of another project, as for instance X window system or the Linux kernel), notabug (the issue is not really a bug), wontfix (the developers consider not to correct this error for any reason) and fixed (the error has been corrected).

- Assigned: Name and/or e-mail address of the developer in charge of fixing this bug.

- Priority: Urgency of the error. It can take following values: immediate, urgent, high, normal and low. Usually this field is modified by the bugmaster as users do not have sufficient knowledge on the software to know the correct value.

- Severity: How this error affects the use and development of the software. Possible values are (from high severity to lower one): blocker, critical, major, normal, minor, trivial and enhancement.

- Reporter: Name and e-mail address of the bug reporter.

- Product: Software that contains the bug. Usually this is given at the tarball level.

- Version: Version number of the product. If no version was introduced, *unspecified* is given. Also, for enhancements the option *unversioned enhancement* may be chosen.

- Component: Minor component of the product.

- Platform: Operating system or architecture where the error appeared.

Usually all fields (besides the automatic ones like *bugid*, the opening date or its status) are filled out the first time by the reporter. Larger projects usually have some professional or volunteer staff that

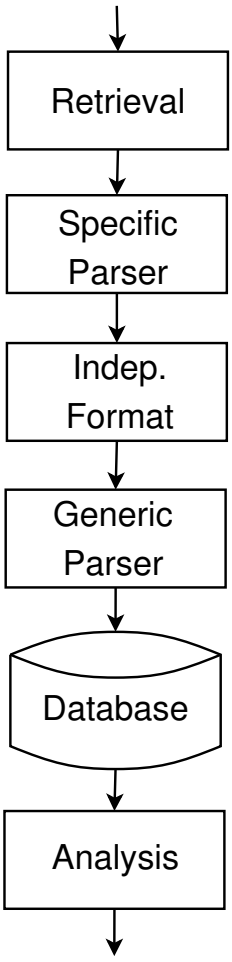


Figure 3.14: *Architecture of the BugZilla Analyzer.*

review the entries in order to adjust the information [Villa, 2003; 2005]. This is especially important for fields like priority or severity as end-users hardly have no knowledge or experience on how to evaluate these fields.

3.5.2 Data acquisition and further processing

For the analysis of the data stored in a bug-tracking system, we have created a tool that is specifically devoted to extract data from BugZilla. The architecture of the BugZilla Analyzing Tool is described in figure 3.14. Although the retrieval of the data could theoretically be simplified by obtaining the database of the BugZilla system from the project administrators, we thought that retrieving the data directly from the web interface would be more in accordance with the non-intrusive policy that all other tools described in this thesis follow.

We had to deal with several problems while retrieving the BugZilla data. After crawling for all web pages (one per bug) and storing them locally, we had to transform the HTML data into an intermediate log-type format, as not all fields were given for all bugs due probably to a transition from a previous system. Probably also because of this, there may have been some information loss and some ids could not be tracked. Other problems that we found, were the existence of wrong date entries for some bugs and comments. As the bug report ids are sequential, we could fix these entries when we found out that the date was wrong. We applied the same solution to comments with erroneous dates, as comments are also posted sequentially and cannot be introduced before the bug report has been submitted.

In recent versions of BugZilla, it is possible to obtain the data in XML format which simplifies in

a great manner the data extraction[19]. When writing this thesis, the use of the XML interface was not as common as the author would wish, so retrieving the data from parsing web pages was the unique non-intrusive manner at that time. In any case, the BugZilla analyzing tool has been designed in such a way that only by removing some parts (specifically the specific HTML-parser which parses into the independent format) and by modifying the generic parser we could reuse the rest of the modules without major changes using the XML query format. This is also valid for other bug-tracking systems, as GNATS.

3.6 Other, project-related sources

Besides the information that can be gathered from widely-used development-supporting and communication tools, other project-specific data sources can be considered. This information includes meta-data or organizational data that the project stores in a possibly structured format. The main problem of these data sources lies in the fact that they are only valid for a project (or sometimes for a small set of projects), so after inspecting the data that a project may provide this analysis has to be customized.

For this work, we have studied in detail information that is provided by the Debian project, that includes information of the packages that are supported by this distribution, data about the popularity of the various packages and the database of maintainers. As noted above, this information is very specific and may not be found for other projects.

3.6.1 Debian Sources File

Since 2.0, the Debian repository contains a Sources.gz file for each release, listing information about every source package in it. For each package, it contains: name and version, list of binary packages built from it, name and e-mail address of the maintainer, and some other information that is not relevant for this study. In some cases, packages are not maintained by individual volunteers, but by teams.

As an example, an excerpt of the entry for the Mozilla source package in Debian 2.2 has been introduced below[20]. It can be seen how it corresponds to version M18-3, provides four binary packages, and is maintained by Frank Belew.

```
[...]
Package: mozilla
Binary: mozilla, mozilla-dev, libnspr4, libnspr4-dev
Version: M18-3
Priority: optional
Section: web
Maintainer: Frank Belew (Myth) <frb@debian.org>
Architecture: any
Standards-Version: 3.2.0
Format: 1.0
Directory: dists/potato/main/source/web
Files:
  57ee230b97ccc69444ccccd0bc66908a 719 mozilla_M18-3.dsc
  532934635ad426255036ee070bad03c8 28642415 mozilla_M18.orig.tar.gz
  3adf83de7e74bf940ee02c0deca20372 18277 mozilla_M18-3.diff.gz
[...]
```

[19]For instance, bug #55,000 from the KDE bug-tracking system, which can be accessed through the web interface at *http : //bugs.kde.org/show*$_b$*ug.cgi?id* = 55000 may also be obtained in XML at following URL: http://bugs.kde.org/xml.cgi?id=55000.

[20]The original Sources.gz file where this entry comes from can be found at http://www.debian.org/mirror/list.

3.6.2 Debian Popularity Contest

The Debian Popularity Contest[21], popularly known as the *PopCon*, is an attempt to map the usage of Debian packages. Its main goal is to know what software packages are actually installed and used. This information may be used, for instance, to decide what packages should compose the first (and most downloaded) Debian CD.

The *PopCon* system functions as follows: Debian users may install the `popcon` package which sends a message every week with the list of packages installed on the machine and the access time of some files that may give a hint of the last usage of these packages. Of course, privacy issues are considered in a number of ways: upon installation, the user is explicitly asked if he wants to send this information to Debian, and the server that collects the data anonymizes it as much as possible.

The resulting statistical information of all users participating in this scheme is publicly available on the web site of the project. For every package it includes the number of machines on which it is installed (inst), the number of machines which make regular use of that package (vote), the number of recent updates (recent), the number of machines where not enough information is available (no-file) and the maintainer of the package. Below is an excerpt of the available data, in this case the top ten packages ordered by installations as of December 4th, 2004.

rank	name	inst	vote	old	recent	no-files	maintainer
1	adduser	6881	6471	94	316	0	Adduser Developers
2	debianutils	6881	6517	50	314	0	Clint Adams
3	diff	6881	6425	261	195	0	Santiago Vila
4	e2fsprogs	6881	5448	825	608	0	Theodore Y. Ts'o
5	findutils	6881	6449	233	199	0	Andreas Metzler
6	grep	6881	6436	126	319	0	Ryan M. Golbeck
7	gzip	6881	6558	245	78	0	Bdale Garbee
8	hostname	6881	6112	715	54	0	Graham Wilson
9	login	6881	6407	56	418	0	Karl Ramm
10	ncurses-base	6881	56	143	6	6676	Daniel Jacobowitz

3.6.3 Debian developer database

From June 1999 onwards, Debian holds a database[22] with data related to members of the project. Since the year 2000, Debian developers should have passed an admission process that assures they know the Debian goals and the guidelines of the project. Some information from the Debian Developer database can be retrieved publicly through the Internet: name, nick or username, e-mail address and PGP/GPG key. In addition, it includes information about the country of residence and the date of entrance to the project (if later than the database creation, else June 20th 1999). We will use in this thesis the data from the Debian Database to study the evolution of the number of developers. In [Robles *et al.*, 2001] a similar study has already been made, including a detailed analysis of the country of residence.

3.7 Integration of different sources

As by now, we have treated all data sources independently. We have seen how to identify a data source, how to retrieve the information it contains, how to extract it and finally how to store it into a database to further analyze it in subsequent steps.

In this section we will see how there are elements that appear in several data sources and how by linking those elements we can have additional (and more complete) information about the development process, the development community and its organization. This type of data integration can be done in three different ways:

[21] The main web page of the Debian Popularity Contest is: http://popcon.debian.org.

[22] The front page of the Debian developer database is: http://db.debian.org.

- Integration through artifacts. We define a granularity level (project, directory, file, class, method or even line) at which we identify all the actions related to every artifact [Antoniol *et al.*, 2005]. For instance, we could link the information from the activity on a file in the versioning repository to the information of its complexity obtained by scanning the source code and calculating any complexity measure.

- Integration by identification of traces from other sources. For instance we find bug report identifiers in version repository logs. Some research groups already have already worked in this direction [Antoniol *et al.*, 2005; Fischer *et al.*, 2003; Mockus *et al.*, 2002; Germán & Mockus, 2003].

- Integration at the developer level. Actors that take part in the development process can be identified, and their activity tracked in the various sources of information, even when different identities are used (several e-mail address, logins, and even different spelling of real names).

In this work, we will focus on the third point, the integration at the developer level. This is an innovative contribution of this thesis, at least to the authors' knowledge. We will analyze how developers appear in several ways through the various data sources and how we can create a structure that links information from all these data sources. In addition, we will discuss how we can obtain extra data once we have aggregated information for developers from several data sources. Finally, we will introduce some ideas of how we want to achieve all this and at the same time preserve the privacy of the affected persons.

3.7.1 Considering developers for data integration

Libre software developers, or more broadly, participants in the creation of libre software (from now on actors) usually interact with one or more Internet-based systems related to the software production and maintenance, some of which have been presented in this chapter and are depicted in Figure 3.15. These systems usually require every actor to adopt an identity to interact with them. The identities actors use are usually different from system to system, and in some cases a given author can have more than one identity for the same system, sometimes successive over time, sometimes even contemporary.

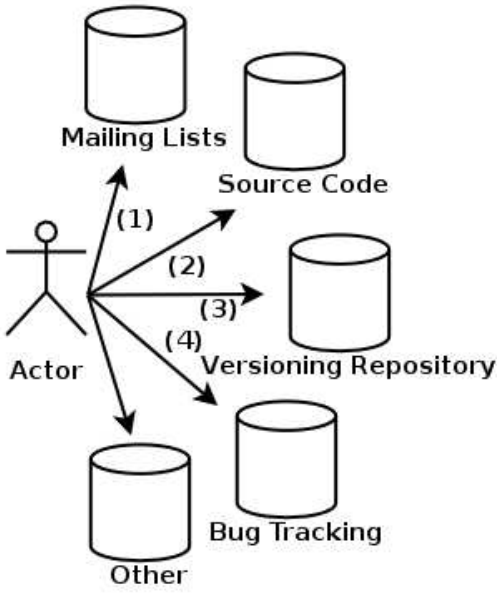


Figure 3.15: *Different systems with which an actor may interact.*

Some kinds of identities are the following (summarized in Table 3.2):

- An actor may post on mailing lists with one or more e-mail addresses (some times linked to a real life name).
- In a source file, an actor can appear with many identities: real life names (such as in copyright notices), e-mail addresses, RCS-type identifiers (such as those automatically maintained by CVS), among others.
- The interaction with the versioning repository occurs through an account in the server machine, which appears in the logs of the system.
- Bug tracking systems require usually to have an account with an associated e-mail address.

Other sources may include entries in weblogs, forums, blogs, and other. Although they are not considered here, the approach proposed could easily include them.

Type	Data Source	Primary Identities
(1)	Mailing lists	username@example.com
(1)	Mailing lists	Name Surname
(2)	Source Code	(c) Name Surname
(2)	Source Code	(c) username@example.com
(2)	Source Code	$id: username$
(3)	Versioning System	username
(4)	Bug Tracking	username@example.com

Table 3.2: *Identities that can be found for each data source.*

Given the various identities linking an actor to his actions on a repository, our goal is to determine all of them that correspond to the same real person. Basically we can classify these identities in two types: primary identities and secondary identities.

- Primary identities are mandatory. For instance, actors need an e-mail address to post a message to a mailing list. Mailing lists, versioning system and bug tracking system require to have at least a mandatory identity in order to participate (although in some exceptional cases this can be done anonymously). Source code does not have primary identities, except in some special projects where the copyright notice or some other authorship information is mandatory.
- Secondary identities are redundant. For instance, actors may provide their real-life name in the e-mails they send, but this is not required. Secondary identities usually appear together with primary identities, and may help in the identification process of actors.

Note that the relationships between actors and repositories have not to be unique: an actor could have one or more different identities in any repository. This is common in mailing lists, where actors may submit from different e-mail addresses, even in the same period of time. But even in cases such as CVS repositories, where such changes seldom happen, an actor may change the username of his account, and of course the same actor could have different usernames in different CVS repositories.

Data fetching, structure and verification

Figure 3.16 shows a glimpse of the data structures that we have designed. The goal with such data structures is to learn the identities from several data sources that correspond to the same person while at the same time preserving his privacy.

The process works as follows: all the identities are introduced into the database in the Identities table. This table is filled by directly extracting identities (using heuristics as shown in this chapter to locate them) from software-related repositories. Besides the identity itself, this table stores identifiers for the repository (data source) where it was found, which could be of value not only in the latter matching process, but also for validation and track-back purposes. The kind of identity (login, e-mail

address, *real name*) is also stored to ease the automatic processing. Hashes of identities are added to provide a mechanism which can be used to deal with privacy issues, as will be described later.

When extracting identities, sometimes relationships among them can be inferred. For instance, a real name can be next to an e-mail address in a *From* field in a message. Those relationships are captured as entries in the Matches table, which will be the center of the matching (identification of identities of the same person) process. The *evidence* field in this table provides insight about every identified match. As the process we are performing is mostly automatic, the value of *evidence* will contain the name of the heuristic that has been used. This will include automatic heuristics, but also human inspection and verification. Sometimes, the information is not enough to ensure that the match is true for sure, and that is the reason why a field showing the estimated probability has been added. Fields that have been verified by humans with absolute certainty will be assigned a probability of 1.

With the information stored in Identities and Matches, the identification process may begin. Unique actors are identified with information in Matches, filling the Identifications table, and choosing unique person identifiers. Other information in the Persons table can be filled directly with data from the repositories or from other sources.

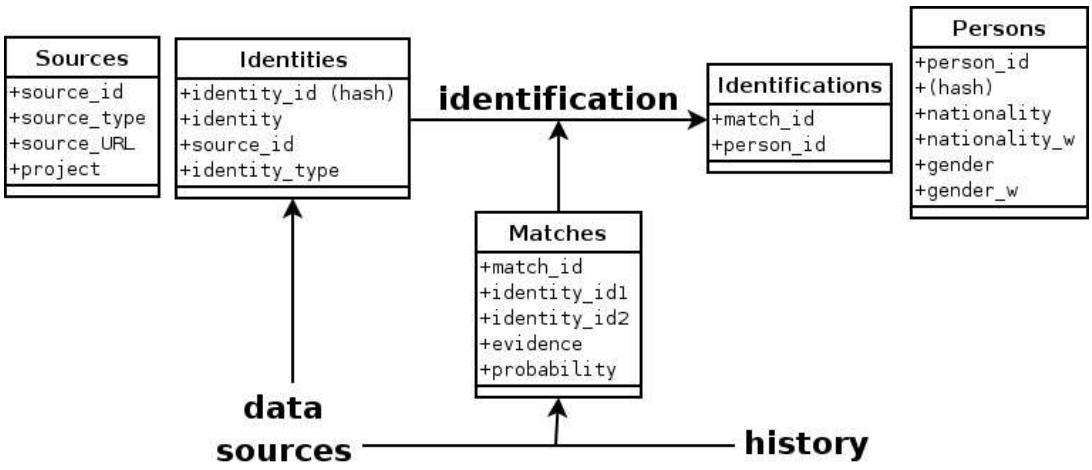


Figure 3.16: *Main tables involved in the matching process and identification of unique actors*

3.7.2 Matching identities more in detail

We will usually have many identities for every actor. For instance, we can have name(s), username(s) and e-mail address(es). Every actor considered will have at least one of them, although possibly he may be identified with several, as is shown in Figure 3.17.

Our problem is how to match all the identities that correspond to the same actor. In other words, we want to fill the Matches table with as much information as possible (and as accurate as possible). As already mentioned this is done using heuristics. Let's discuss some of them with some detail:

- In many cases it is common to find a secondary identity associated to a primary one. This happens often in mailing lists, source code authorship assignments and bug tracking system. In all these cases, the primary identity (usually an e-mail address) may have a *real life* name associated to it. Consider, for instance, *Example User <username@example.com>*, which implies that *Example User* and *<username@example.com>* correspond to the same actor. GPG key rings can also be a useful source of matches. A GPG key contain a list of e-mail addresses that a given person may use for encryption and authentication purposes. GPG is very popular in the libre software community and there exist GPG servers that store GPG keys with all these information.

- Sometimes an identity can be constructed from another one. For instance, the *real life* name can be extracted in some cases from the e-mail username. Many e-mail addresses follow a given

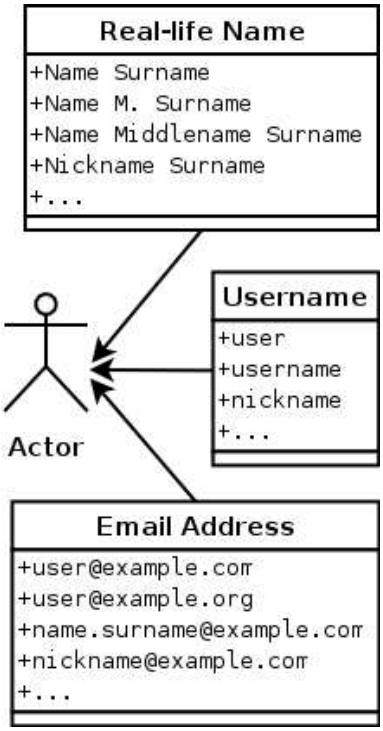


Figure 3.17: *An actor with three different kinds of identities.*

structure, such as *name.surname@example.com* or *name_surname@example.com*. We can easily infer in those cases the *real life* name of the actor. Other, more complex, routines may be used for extracting names from e-mails, with procedures from the machine learning world as for instance applying *named entity recognition* [Minkov *et al.*, 2005].

- In many cases one identity is a part of some other. For instance, it is common that the username obtained from CVS is the same as the username part of the e-mail address. This can be matched automatically, and lated verified by other means. This is one of the more error-prone heuristics, and is of course not useful for very popular usernames like *joe*. But despite these facts, it has proven to be very useful.

- Some projects or repositories maintain specific information that can be used for matching, for instance, because a list of contributors is maintained. As an example, the KDE project has a file in the CVS repository which lists, for every person with write access to the CVS, his *real life* name, his username for CVS and an e-mail address[23]. Other similar case are developers registered in the SourceForge platform, who have a personal page where they may include their *real life* name.

Of course this is not an exhaustive list, and combinations of the described heuristics can be used. For instance, a mixed approach could benefit from the data in changelog files [Capiluppi *et al.*, 2003] for finding identity matches.

Usually, the fraction of false positives for matches can be minimized by taking into account the project from which the data was obtained. If we have a *joe* entry as username for the CVS repository in an specific project, and in that same project we find somebody whose e-mail address is joe@example.com (and no other e-mail address that could be suspicious of being from a *joe*) then there is a high probability that both are identities of the same actor.

In any case, the fraction of false positives will never be zero for large quantities of identities. Therefore, some heuristics are specifically designed for cleaning the Matches table (eliminating those

[23]The list that the KDE project offers can be obtained from: http://websvn.kde.org/trunk/KDE/kde-common/accounts?rev=480789

entries which are not correct, despite being found by an heuristic) and verification, including human verification. In some cases, the help from an expert that knows about the membership of a project, for instance, could be of great help.

But even after cleaning and verification, some matches will be false, and some will be missing, which can cause problems. However, since we are interested in using the collected data for statistical purposes, this should not be a big issue provided the error rate is small enough.

3.7.3 Privacy issues

Privacy is of course an important concern when dealing with sensible data like this. Although all the information used is public, and contains almost no private data (phone, physical address, age, wage...), the quantity and detail of the information that is available for any single developer can cause privacy problems. Therefore, we have devised a data schema which allows both for the careful control of who has access to linking data to identified real persons, and for the distribution of information preserving anonymity. In the latter case, the information can be distributed in such a way that real persons are not directly identifiable, but new data sets can be, however, combined with the distributed one. This will for sure allow for a safe exchange of information between research groups.

For this purpose, the hashes of identities serve as a firewall. They are easy to compute from real identities, but are not useful for recovering them when only the hashes are available. Therefore, the Matches, Identifications and Persons tables can be distributed without compromising the real identities of developers as a whole. However, new data sets can be combined. Assuming a research group has a similar schema (or uses the same one), with some identities found, the corresponding hash can be calculated for any of them and it may be looked up in the Matches table. Of course this will not be useful in many cases for finding new matches, but it would always allow to link an identity (and the data associated with it) to an actor in the Persons table. Therefore, any development data distributed using hash identities instead of developer names can be safely shared (but see below).

Although hashes will make it impossible to track real persons from the distributed data, it is still possible to look for certain persons in the data set. By hashing the usual identifiers of those persons, they can be found in the Matches table, and their identity is thus discovered. That is the reason why although distributing hashes to other research groups under reasonable ethical agreements is acceptable, probably it is not to do the same for anyone.

To avoid this problem, our schema has still a second level of privacy firewall: the person identifier in the Persons table. This identifier is given in such a way that it cannot be used in any way to infer the identities of an actor without having access to the Identifications table. Therefore it is enough to key all development data with this person identifiers, and distributing only the Persons table in addition to that data to ensure the full privacy of the involved developers.

Of course, even in this latter case somebody could go to the software repositories used to obtain the data, and try to match the results with the distributed information. But this is an unavoidable problem: a third party can always *milk* the same repositories, and obtain exactly the same data, including real identities. In fact, this is the basis of the reproducibility of the studies.

3.7.4 Automatic (post-identification) analysis

The reader has probably noted that the Persons table in Figure 3.16 includes some fields with personal information. We have devised some heuristics to infer some of them from data in the repositories, usually from the structure of identities. For instance, nationality can be guessed by several means:

- Analyzing the top level domain (TLD) of the various e-mail addresses found in the identities could be a first possibility. The algorithm in this case consists of listing all e-mail addresses, extracting the TLD from them, rejecting those TLD that cannot be directly assigned to a country (.com, .net, .org, and other) or those who are from *fake* countries (.nu, and other), and finally looking at the remaining TLDs and count how often they occur. The TLD that is more frequent gives a hint about the nationality of the person. Of course this heuristic is especially bad for US-based actors (since they are not likely to use the US TLD), and for those using .org or .com addresses, quite common in libre software projects.

- From the analysis of the mailing lists we could extract the time zone from the *Date* header. This information is very inaccurate on its own, but combined with the previous point it has shown to be very powerful. In this sense, it will allow us to filter those TLDs not assigned to countries and support other assignations.
- Another approach is to use whois data for the second level domain in e-mail address, considering that the whois contact information (which includes a physical mail address) is valid as an estimator of the country of the actor. Of course, this is not always the case.

Other case example of information which can be obtained from identities is the gender. Usually we can infer the gender from the name of the person. However, in some cases it depends on the nationality, since some names may be assigned to males in one country and to females in another. This is for instance the case for Andrea, which in Italy is a male name while in Germany, Spain and other countries is given usually to females.

Chapter 4

Methodologies and Analyses

> *There is in the universe something for the description and analysis of which the natural sciences cannot contribute anything. There are events beyond the range of those events that the procedures of the natural sciences are fit to observe and describe. There is human action.*
> Ludwig von Mises

This chapter presents some methodological and analytical procedures that can be applied on publicly obtained data sources from libre software projects. It builds on top of the data that has been extracted from data sources described in detail in chapter 3. The analyses and methodologies that will be presented give an ample perspective of the libre software phenomenon, especially of large and well-known libre software projects.

The chapter starts with a reproduction of a *classical* analysis, the study of the evolution for large libre software applications. The aim of this study is to find out if the *laws* of software evolution are valid for these type of projects. The second analysis is related to the evolution of software compilations, which group thousands of libre software applications. We have selected the Debian project and have observed how it has evolved for the last seven years. In addition, we study the amount of code that remains from past states of the software for a large set of libre software projects. This procedure has been labeled as *software archaeology*.

All analyses up to this point, including software archaeology, are centered on *classical* source code (those files written in a programming language). But along with *classical* source code, in software projects other type of sources are contained. We propose a file-type based analysis to gain more knowledge on them. This introduces some social issues, which we will comment in depth by means of a software network analysis. This methodology shows that the idea that software is done by isolated *hackers* without interactions with others is one of the biggest fallacies of the libre software phenomenon, at least in regard to large projects. Libre software is developed in a community, a social body that has a structure that flexible and self-organized. By means of a social network analysis we want to infer the relationships and dependencies that exist among the entities (entities can be developers or software applications) and deduce interesting information from the position that entities have in the resulting network.

Once we have studied the social structure of some projects, we will try to detect if this structure is static or if it evolves. In detail, the most important fact to be researched is if the leading developer group changes its composition over the years. A possibility is that libre software projects are led by several generations; another one, that the *core* remains stable. Our analysis yields that we can find generations of developers entering and leaving the *core*. The next question is then to see how the integration process of new developers is (see section 4.8).

4.1 Classification of the analyses

In this section we present a classification of the analyses proposed in this chapter. The goal of the classification is to to provide further insight about the nature of the various analyses. The classification

is based on the data sources that are being used, the properties of the analyses and the projects that have been used as case study.

4.1.1 Data sources

Chapter 3 contained a detailed description of the data sources found for libre software projects. The ones that have been used for the methodologies and analyses performed in this chapter are shown in table 4.1. All data sources have been used at least once, and some of them, especially source code and meta-data from the versioning system, appear more frequently.

Methodology / Analysis	Source Code	Versioning system	Mailing lists	Bug-tracking system	Other, project-related	Integration
Software Evolution	X					
Evolution of Compilations	X				X	
Software Archaeology		X				
File-type based Analysis		X				
Social Network Analysis	X	X				
Developer Generations		X			X	
Developer Integration		X	X	X		X

Table 4.1: *Data sources used in the analyses.*

The software evolution analysis only requires having source code, although it is necessary to have it for several points in time. For the evolution of software compilations we have used source code and needed other information sources to know the software packages that have to be retrieved. For Debian, our case study, this means making use of the Sources.gz files (see subsection 3.6.1).

Software archaeology only requires meta-data from versioning systems, to be obtained using the DrJones tool (presented in subsection 3.3.3). The file-type based analysis is in the same situation: we only require the logs of the versioning system which contain enough information about the files and commiters involved.

The social network analysis uses data extracted from source code files and from the interactions recorded in the versioning system. The study of generations makes use of data from the versioning system and project-related data from the Debian project (Sources.gz described in subsection 3.6.1, the Popularity Contest presented in subsection 3.6.2 and the Debian Developer database depicted in subsection 3.6.3).

Finally, the part devoted to the integration of new developers to the core group of a project is based on developer traces found in the versioning system, the mailing lists and the bug-tracking system. Integration of data obtained from different sources as described in section 3.7 is of great importance for this methodology.

4.1.2 Characteristics of the analyses

Regarding the nature of the analyses, we have defined a set of characteristics that will help situating the analysis presented in this thesis. So, we look if analysis are technically oriented (their scope is targeted towards technical artifacts), have a social background (devoted more specifically to developers and relationships), if they are longitudinal (time is considered as a factor), if they are based on public available data and finally its granularity (i.e. if the scope of the study is a project by itself or the aggregation of multiple projects). Table 4.2 gives the result of classifying our analyses according to these criteria.

Methodology / Analysis	Technological	Social	Longitudinal analysis	Public Data set	Granularity
Software Evolution	X		X	X	M
Evolution of Compilations	X		X	X	D
Software Archaeology	X	(X)	(X)	X	M
File-type based Analysis	X	X	X	X	R
Social Network Analysis		X	X	X	R
Developer Generations		X	X	X	M
Developer Integration		X	X	X	R

Table 4.2: *Characteristics of the analyses.*

We start with the purely technological analyses (software evolution and evolution of software compilations) and turn afterwards to social analyses. Software archaeology and the file-type based analysis have characteristics of both, although the former has a more technical scope. The latter gives an idea of the social groups that form the project and how they are related to technical issues. Finally, the social network analysis, the core group generation and the integration of new developers are mainly concerned with the human resources of projects. In any case, technical and social structure have many links in common and one cannot be understood without the other. In our case studies, taken from the libre software world, both views are of great important as there no pre-defined organizational structure exists. Technical and social hierarchies are created dynamically and evolve constantly.

A characteristic that is given in all the analyses is the fact that they all take time into account. This links with the previous idea of a flexible environment where evolution is the key. Maybe the case of software archaeology is the one where the longitudinal analysis is not that clear; strictly, performing a longitudinal analysis of software archaeology would imply to repeat the analysis for several points in time. This is not done in this thesis, although it could be seen as a good idea for future research. But as archaeology is an analysis where time is embedded, even if not purely longitudinal it can be seen as a type of longitudinal analysis (thus, we have marked it that way in table 4.2).

The other characteristic that is shared among all the analyses is that data sources are publicly available. In subsection 4.1.3 we introduce several projects chosen as case studies; all of them (and information associated to them) can be accessed freely through the Internet. If our methodologies have the property of being repeatable, the availability of the data assures that our studies can be reproduced with the same data sets used in this thesis.

Finally, not all analyses have been performed on the same projects for a set of reasons. The type of analysis depends on the *granularity* of the project, meaning by granularity the size of a project as well as its interrelation with other projects. In this sense, we have to face some concepts which are loosely defined in the libre software world such as *project* or *community*. There exists no convention or definition that refers to GNOME as a project, and to Evolution (a groupware application part of GNOME) as one of its subprojects or if, on the other hand, Evolution is a project per se and GNOME its super-project. This point has special importance for our analyses as some of them are especially suited for certain project sizes. Hence, we have defined here some levels of granularity and will from now on keep on with this nomenclature although noticing in advance that it is not standard.

Figure 4.1 depicts how we have defined the various terms. We use the way CVS versioning system are organized, modules and repositories, and aggregate distributions to that least. The reason is twofold: first, many of our analyses are based on CVS data as shown in table 4.1, so this will allow to just take the concepts from there and use them *ad-hoc*. Second, it is clear enough and easy to define for our purposes, so no further depth in our classification is needed.

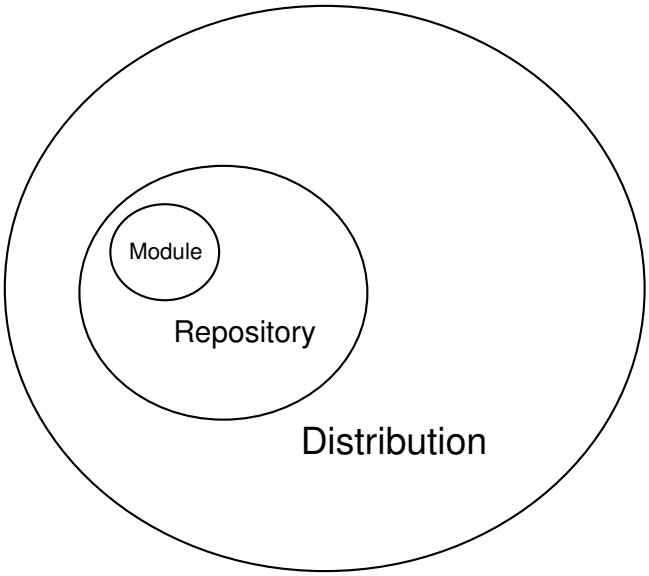


Figure 4.1: *Different kinds of (technical and social) granularity.*

The lowest level of granularity for our analysis is at the module level. It should be noted that in this case we are at the level of CVS modules, which should be considered as a high-level container (a directory) of a software and not as a source file as for instance Lehman et al. [Lehman & Belady, 1985] understand it in their *classical* software evolution analyses. A module in our case should be seen as a complete application or library. This means that we are referring to modules with Evolution, GNOME's personal information management tool, kdelibs, a library for KDE, or the Apache web server, part of the Apache project.

The next level, which comprises a set of modules that have technical or organizational dependencies among them, is the repository. Some example of repositories could be the GNOME project (which in addition to Evolution, contains the Nautilus file manager, the Galeon web browser and some hundreds of modules more), the KDE project or the Apache project (notice that here we include in addition to the web server all other modules in its CVS repository as for instance those that belong to Jakarta).

Finally, the largest granularity level considered in this thesis are *distributions* which are composed of many repositories. Usually, distributions work on a different scale than repositories and modules, as their primary task is to create an integrated body of software which is easy to install and manage. Theirs is not purely development work and, with some exceptions, they do not provide over a versioning system where we could track all the changes that they apply. On the contrary, usually distributions tightly work with the original authors at the repository or more generally at the module level. Examples of distributions are Debian (the one considered in this thesis), Red Hat, SuSE, or Ubuntu.

Methodology / Analysis	Libre software projects used as case studies
Software Evolution	Linux, FreeBSD kernel, OpenBSD kernel, NetBSD kernel, kdelibs, jakarta-commons, mcs, mono, KOffice, kdepim, Gnumeric, GTK+, xml-xerces, Galeon, httpd-2.0, xml-xalan, kdebase, kdenetwork, KDevelop, ant, Evolution and The GIMP
Evolution of Compilations	Debian GNU/Linux (from version 2.0 to 3.1 "Sarge")
Software Archaeology	GNU Emacs, GCC, Wine, GTK+, The GIMP, Apache-1.3, kdelibs, Evolution, Mozilla
File-type based Analysis	KDE
Social Network Analysis	Linux, Apache, GNOME, KDE
Developer Generations	The GIMP, Mozilla, Evolution, FreeBSD, kdelibs, jakarta-commons, mcs, mono, KOffice, kdepim, Gnumeric, GTK+, Galeon, xml-xalan, kdebase, kdenetwork, KDevelop, ant and Nautilus
Developer Integration	GNOME

Table 4.3: *Projects used as case studies for the various analyses.*

The last column in table 4.2 gives the granularity level for each analysis. M is assigned to those that have been performed at the module level, R to those at the repository level and D to he ones at the distribution level.

4.1.3 Projects selected as case studies

The project that is studied at the distribution level is the Debian Linux-based distribution.

At the repository level, the projects that have been selected as case studies in this thesis are following: GNOME, KDE, Apache, Linux, *BSD (FreeBSD, OpenBSD, NetBSD), Mono and Mozilla. All of them have a libre software license and an ample community surrounding them (ie. contributors can be counted in the thousands for any of them). This kind of libre software projects are also large in size (at least in the order of 100K lines of code). This means that the technical complexity they have to cope is not insignificant. Table 4.3 contains the details of the projects used for the various analyses. It should be noted, that depending on the granularity of the analysis, as shown in the last subsection, modules, repositories or even distributions may appear.

Finally, a large number of applications (the module level) have been selected as case studies. Most of them are part of larger projects (belong to any of the repositories cited before).

For a specific introduction to any of these projects, see appendix A.

4.2 Software Evolution

As already broadly introduced in subsection 2.3.2, thirty years of research on software evolution have resulted in a set of *laws*, known as Lehman's Laws of Software Evolution [Lehman & Belady, 1985; Lehman & Ramil, 2001]. The number of laws has grown from three in the seventies to eight in their latest version [Lehman *et al.*, 1997] and all of them have been empirically validated by studying projects developed in traditional industrial software development environments.

4.2.1 Goals

Our intention is to explore how some large libre projects behave in the context of the laws of software evolution, especially regarding to software growth. For this matter, we start by reproducing (with current data) the study performed five years ago on the Linux kernel [Godfrey & Tu, 2000] (see subsection 2.3.2), which questions the conformance of libre software projects to some of those laws.

In addition, we have extended the study to other libre software systems in the same domain (operating system kernels): the *BSD family. Finally, we have performed it on 18 other large libre software applications with the intention of finding if we can identify give general patterns in the results.

In all these cases, we also have another goal in mind: to find differences in the evolution of the software before and after version 1.0 is released. Traditional software evolution studies consider only releases after the first one delivered to customers (usually this software gets the version number 1.0 and is considered to be *stable*). However, it is a common behavior in many libre software project to follow the "release early, release often" *rule*, which means that programs are available to the public well before they are considered stable. In other words, we consider the first release named *stable* is not that special. Therefore, it is difficult to find a point where *development* finishes and *evolution* starts. We have studied the applications from their first releases (if available) onwards (even if these first releases were before the 1.0 release) with the hope of finding the significance of the 1.0 release from our results.

4.2.2 Methodology

The methodology used in this thesis is based on analyzing publicly available source code on the Internet. The code corresponding to every snapshot considered is downloaded to a local directory, where its size is computed. The results are stored in a database, which is later used for performing a detailed analysis and for plotting the graphs. The size of each snapshot is obtained by using the SLOCCount tool presented in section 2.2.3, so we count only source lines of code (SLOC) written in identified source code files.

Depending on the project, we have retrieved the sources in a different way. In the case of the Linux kernel, there is no public CVS versioning repository. Therefore we have decided to download release packages, which are available in Linux mirrors[1]. We have retrieved the official and experimental kernel releases, from 1.0 to the last one published in December 2004 (2.6.10). We have also measured those known as *historic* releases, that is, those released prior to version 1.0 (which dates from March 1994). In order to recreate Godfrey's study on the Linux subsystems, we have gathered data from all main subdirectories (which we call subsystems from now on) in addition to compute the number of source lines of code for the whole system.

For all other systems considered in this study, public CVS repositories are available. In libre software projects, it is common practice that even if *unstable*, the software in the repository can be compiled and is usable, up to the point that automatically generated nightly-builds are offered in many cases, i.e. it is software that is not in a state of flux. Releasing a new version of the software consists usually of taking one of this snapshots and assigning it a specific release name/number, although some projects have more sophisticated procedures as described by Ehrenkrantz et al. (see section 2.3.2 in related research) [Ehrenkranzt, 2003].

Taking these facts into account, we have retrieved monthly snapshots from the CVS repositories, starting by the time the repository was established, until April 2005. The whole process has been

[1] A list of the Linux kernel mirrors can be found at http://www.kernel.org.

automated with the GlueTheos tool (described in subsection 3.2.3). For the *BSD projects, only the operating system kernels (directory src/sys) are considered, while for the rest the whole CVS module is studied.

Although Lehman suggests plotting software size against release numbers, we have done it against time because of two reasons. First, we feel this way matches better the semi-continuous release process found in many libre software projects. And second, this way of depicting evolution is also the one that has been used by Godfrey & Tu for Linux. This implies that using periodic CVS snapshots is enough for our purposes, and we do not need the source packages for specific releases.

Another sensible difference with Lehman's studies lies in the metric used for software size. Lehman uses source code files (which he called "modules" not to be confound with the definition of module used in this thesis), because he argues this metric is more consistent (it has a "higher degree of *semantic integrity*") than considering source lines of code [Lehman *et al.*, 2001]. Godfrey and Tu counted uncommented lines of code, and we do the same. However, preliminary studies on the Linux kernel yield that the mean size of modules (counted in lines of code) remains almost constant over the years; we have observed the same behavior in a fast inspection for some projects. This would imply that counting number of lines or number of source code files would give the same evolution patterns. Further research is needed, however, to verify if this is valid in general.

As a final note, it is worth mentioning that in some cases the projects under study started outside a CVS versioning system, but were later uploaded to one. In those cases, an initial gap will appear in the plots.

4.2.3 Observations on the Linux kernel

Table 4.4 exhibits the main differences between the analysis on Linux by Godfrey and Tu, and the one that is presented in this thesis. When the first study was performed (in the year 2000), two concurrent Linux versions existed: the stable version 2.2.14 and the development version 2.3.39. 34 of the 67 stable releases, and 62 of the 369 development releases were analyzed in it, totaling 96 releases. In our study, we have considered all the releases published, both stable and development, including those prior to the considered first stable release (1.0), and up to 2.6.10 (released December 24th 2004). All in all, we have studied 123 stable and 457 development releases. It should be noted that even if the 2.6 branch is the bleeding-edge stable branch, previous stable branches (2.0, 2.2 and 2.4) are still actively maintained (although usually without the addition of new functionality, as only bugs are removed) and new releases appear from time to time. The number of lines of code in 2.6.10 surpasses the 4 millions, with a tarball size of about 45.5 MB. These figures can be compared to those of 2.3.39: about 1.5 million lines of code, and a tarball size of about 17 MB.

	Godfrey & Tu	Our study
Date	January 2000	December 2004
# of releases	369 D + 67 S	457 D + 123 S
Studied releases	62 D + 34 S	457 D + 123 S
Most recent ver.	2.2.14 (2.3.39)	2.6.10
Size most recent	1,425K (1,607K)	4,147K
Tarball size (MB)	15.9 (17.7)	45.5
Counting	wc + awk	SLOCCount

Table 4.4: *Comparison between Godfrey and Tu's and our study.*

SLOCCount identifies the programming languages in which the analyzed software is written in. We have found that the topmost used language in the Linux kernel is, as expected, C. Figure 4.2 shows two pies with the proportion of C versus other languages for versions 1.0 and 2.6.10 respectively. In both releases, C accounts for more than 90% of the files counted, and for more than 95% of the total SLOCs. The next languages in importance are assembler and shell scripts.

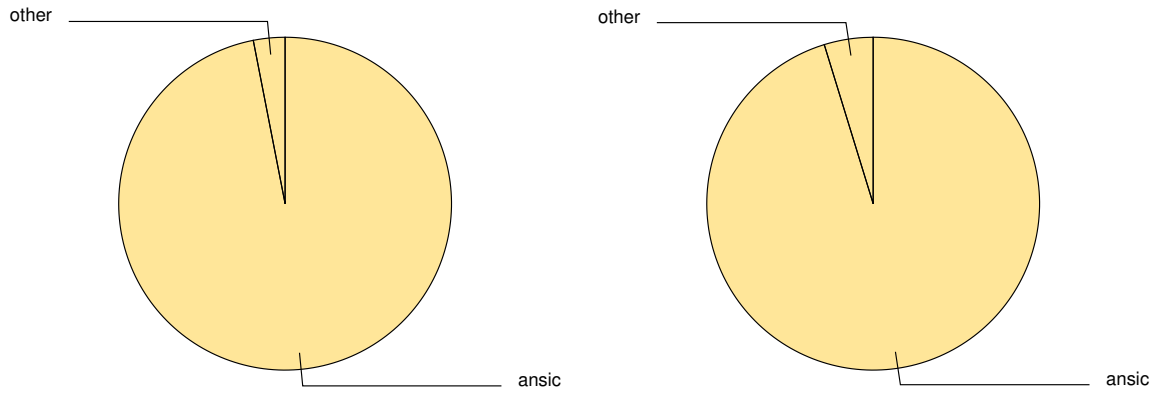


Figure 4.2: *Right: Language distribution for Linux 1.0. Left: Language distribution for Linux 2.6.10.*

System level growth for Linux

Figure 4.3 displays the growth of Linux in terms of sources lines of code from the first versions in 1991 to the most recent one in 2005. We have depicted two vertical lines in all figures for Linux that identify the release date of the 1.0 version (in the year 1994) and the time of the study by Godfrey and Tu (in the year 2000). It can be noted that the super-linearity that was found by Godfrey and Tu seems at first glance to have become more remarkable with time. Based on statistical analysis we have obtained the following software growth equation:

$$y = 0.26 \ \cdot t^2 - 322 \ \cdot t + 195,183 \tag{4.1}$$

where y is the size in source lines of code and t the number of days since version 1.0. The coefficient of determination computed using least squares is $r^2 = .990$. Similarities arise when compared to Godfrey and Tu's equation from 2000 [Godfrey & Tu, 2001]:

$$y = 0.21 \ \cdot t^2 + 252 \ \cdot t + 90,055 \tag{4.2}$$

This supports our initial perception in which we assumed that the super-linear growth has become more remarkably over time. This is demonstrated by the fact that the factor that multiplies the quadratical growth is now 0.26 instead of 0.21, meaning that the growth of Linux has accelerated during the last five years.

Another fact that can be observed from this figure is that the growth pattern followed by the Linux developers has changed over time. Until the year 2000 we find a strong growth in the development branches, with stable branches emerging from it with almost horizontal shapes (stagnation), while the newer stable branches 2.4 and 2.6 show steep growths. That way, the current stable branch (2.6) is growing steadily, with a profile similar to a development branch. However, the latest releases seem to slow down, maybe foreseeing the stabilization that occurs just before the start of a new development branch (which would be 2.7)[2].

Figure 4.4 is composed of two plots: one on the left which describes the growth (in bytes) of the tarball sizes in bytes for all considered Linux releases and another one on the right which plots the number of files over time for Linux. With slight differences, we can see that their shape is similar to the one that we have observed for the growth in terms of SLOC in figure 4.3.

Growth of major subsystems

Godfrey and Tu included in their study the analysis of the major Linux subsystems, following an idea by Gall et al. who stated that looking at the evolution of subsystems would bring more insight about

[2]In fact, Linus Torvalds treats the 2.6 branch as the only branch, avoiding thus the traditional stable-development branches that are developed or maintained in parallel.

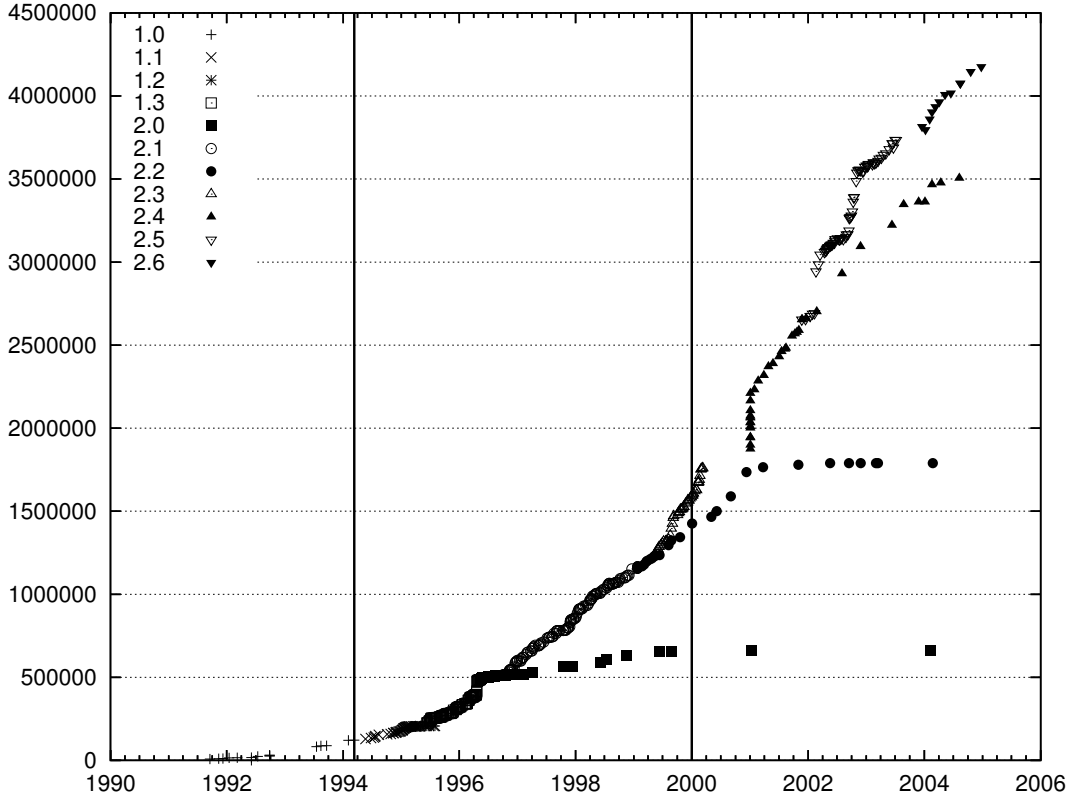


Figure 4.3: *Growth (lines of code) of Linux. The vertical axis is given in SLOC, while the horizontal axis gives the time. The shape of the points in the curve depends on the Linux branch (see legend).*

the software under consideration [Gall *et al.*, 1997].

The growth of major subsystems can be seen in figure 4.5. As in the original work, the most growing subsystem is the one comprising `drivers`, which grows steadily even though in version 2.5.x the `sound` subsystem has been taken apart (which justifies the ripple around 2002). The `drivers` subsystem is followed in the distance by `arch`, `include`, `fs`, recently `sound` and `net`.

If we filter out the `drivers` subsystem from figure 4.5, we can identify the `arch`, `fs`, `include` and `net` subsystems and observe that their growth show super-linear patterns. This occurs even in the case of `net`, which has a growth that is not that steep. Hence, we can conclude that super-linear growth patterns in Linux can also be found at the subsystem level.

The rest of subsystems is hardly visible on figure 4.5 because of their small size. Therefore we have considered them in an extra figure (see figure 4.6) with an y-axis that goes up to 20,000 lines of source code. Even if the size of these subsystems is relatively small, some of them may be considered the core of the Linux project (for instance, the `kernel` subsystem) or comprise functionality that performs crucial tasks (for instance `init` or the `mm` memory management subsystems). The behaviour before 1994 (the release date for version 1.0) for the *kernel* subsystem is chaotic and much of its code has been later moved to other subsystems. Since 1995 it shows a super-linear growth almost equal to the one exhibited by the *mm* subsystem. The rest of the subsystems do not have clear growth patterns: besides *lib*, which has recently started to grow, the rest remain almost constant for a long time, although from time to time they are affected by small gaps because of addition of external code or removal of some code fragments.

Performing a statistical fit on the data at the subsystem level, we obtain the growth equations listed in table 4.5. Subsystems are ordered by their values of r^2, being those with an equation that fits the curve better at the top. We can see that up to the 6th position (which is occupied by the `fs` subsystem) the quadratic fit has acceptable values of r^2, while the last four do not fall in that category. Interestingly enough, these last four correspond all of them to subsystems which are small in size (they all belong to figure 4.6). We assume that this is because of following reason: these modules contain

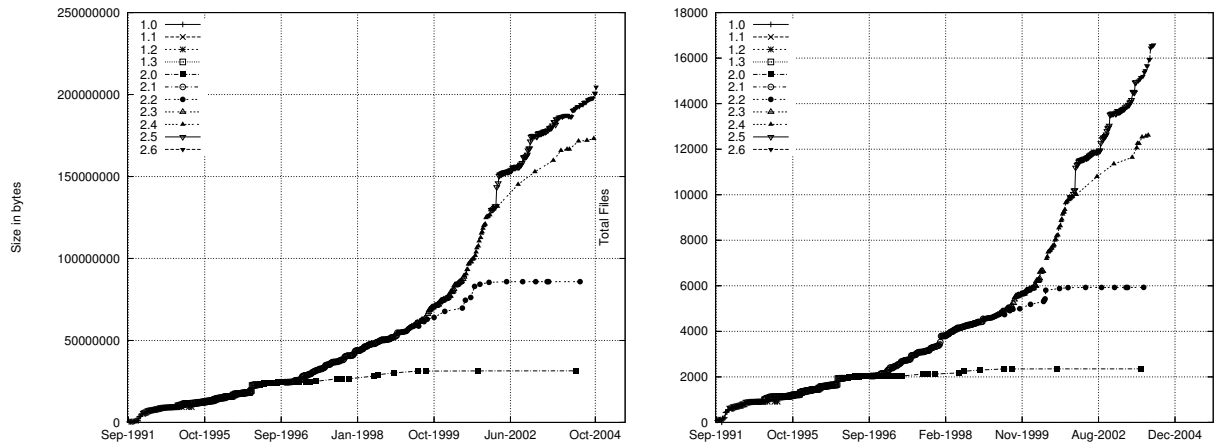


Figure 4.4: *Right: Growth of the tar file for the full Linux kernel source release. Left: Growth in the number of files in full Linux kernel source release.*

crucial functionality for the proper functioning of the kernel and the reliance of the whole depends more heavily on these modules than on the others. So, a driver may have errors, but even if this may be unpleasant it would not affect the rest of the system. This is not the case for an error in the `kernel` subsystem.

Subsystem	Growth equation (statistical fit)	r^2
drivers	$y = 0.06 \cdot t^2 + 336.1 - 70916.4$	.989
net	$y = 0.008 \cdot t^2 + 23.8 \cdot t - 9963.8$	.988
include	$y = 0.02 \cdot t^2 - 7.3 \cdot t - 2528.1$	.983
arch	$y = 0.04 \cdot t^2 + 77.7 + 4432.3$	.980
mm	$y = 0.0005 \cdot t^2 + 0.7 \cdot t + 1649.6$	.977
fs	$y = 0.03 \cdot t^2 - 54.7 \cdot t + 51331.7$	.958
lib	$y = 0.0006 \cdot t^2 - 1.6 \cdot t + 1522.0$	.893
kernel	$y = 0.002 \cdot t^2 - 6.2 + 10471.7399$	.759
ipc	$y = 0.0001 \cdot t^2 + 0.2 \cdot t + 1573.8$	.675
init	$y = 0.0001 \cdot t^2 - 0.01 \cdot t + 450.8$	.668

Table 4.5: *Growth equation for all major Linux subsystems (based on statistical analysis).*

Relative size at the subsystem level

Figure 4.7 displays the evolution of the relative size of the major subsystems. Godfrey and Tu plotted this graph from version 1.0 (in 1994) onwards, while we depict also the previous versions. Those early releases show an erratic behavior, because the architecture for Linux was at that time not specified and changed drastically several times. This can be especially observed for the `kernel` subsystem. But from version 1.0 onwards, all major subsystems have an almost parallel growth pattern, meaning that their relative growth is similar. Again, the gap that can be found in early 2002 for drivers is due to the removal of *sound*, while the one in early 2001 is due to some code being allocated to the *arch* subsystem, as it can be clearly observed in the figure.

Besides these inconsistencies, we can observe how the share of code corresponding to the *drivers* subsystem has remained almost constant since 2000, while in the period from 1998 to 2000 its share grew from around 50% to 60% of the total kernel size, even when Linux itself doubled its size during that period. On the other hand, the share of *net* and *fs* decreases, although it seems that in the latter case its presence has remained around 10% during the last seven years, while *include* and *sound* remain almost constant through the whole system life, since version 1.0.

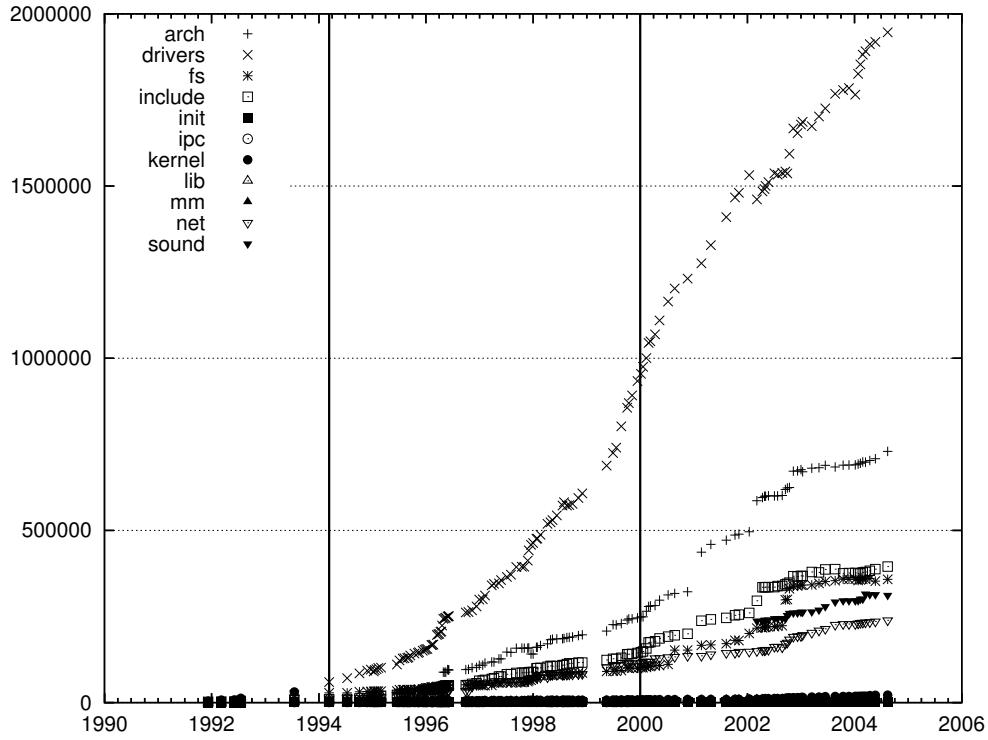


Figure 4.5: *Growth of the major subsystems in Linux (only development releases). The vertical axis is given in SLOC. The horizontal axis gives time.*

Growth of the drivers subsystem

Performing our study with on a smaller granularity level, we can study the subsystems that compose the *driver* subsystem, which is by far the most important in terms of lines of code. For almost all those subsystems we can see no super-linear growth (details can be found in figure 4.8). The only curve that has such a behaviour is the one that groups the *rest* of the drivers (those that are not one of the major ones). The rest of the drivers show linear trends with periods of high activity, when a lot of code is included at once (as for instance in early 2004 for *scsi*), or regularly (the *net* subsystem grows in a pronounced way from mid 1999 to early 2000, clearly different from the linear trend before and after that period). It can be observed that the sum of *other* surpasses it. But none of the subsystems in *other* is larger than *net*; the reason for its growth is that the number of drivers included as `other` has raised to 37.

In general, and in all subsystems, when we perform a study on a smaller granularity level, super-linearity gets less and less frequent while linearity arises. Godfrey and Tu pointed out the existence of independent development groups that worked in parallel due to the high modularization of the Linux kernel. Answering the question if the linear growth patterns that arise at a detailed level of analysis correspond to these development groups could be a promising line for future research. This could mean that each subsystem would behave as a whole, independent system, with its own (linear) growth pattern, being the one for the whole kernel just the addition of those independent behaviors and hence yielding even a super-linear pattern as a trend towards the raise in the number of subsystems can be observed.

Figure 4.9 displays on the left the growth for all the other drivers subsystems that had been grouped in a previous figure as *drivers/other*. The large number of them makes the plot a little bit chaotic, but sufficiently clear to explain that in general these drivers contain few code (under 20,000 SLOC) and that the ones who are larger have erratic behaviors. Probably this is due to the nature of the drivers, which can be built independently from each other and have been most probably developed by different developer groups. Drivers have large times of validity, because they are usually not removed with newer versions of Linux even if the hardware they support is old.

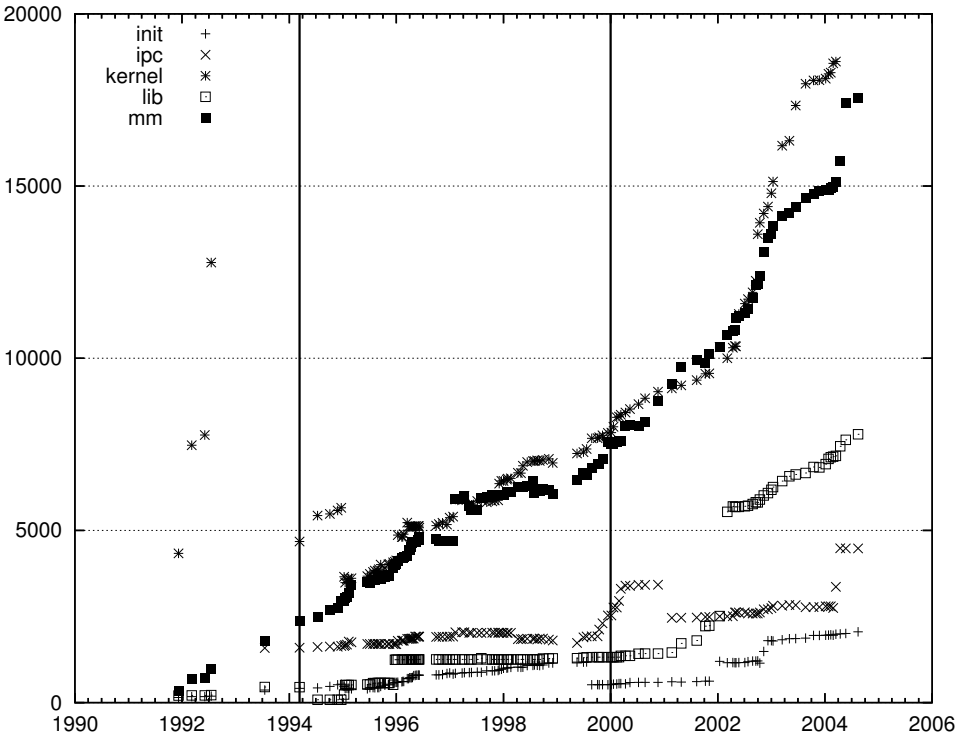


Figure 4.6: *Growth of the smaller, core subsystems in Linux (development releases). Vertical axis is in SLOC. The horizontal axis gives time.*

On the right of figure 4.9 we have depicted the subsystems of the `arch` subsystem. We may observe here a similar behaviour as the one found for the drivers subsystems at a smaller level (in this case the top subsystems have no more than 75,000 lines of code) and with the emergence of many gaps which are probably due to the inclusion of large amounts of code authored by third parties. In this sense, it is more difficult to argue for work done in parallel by several groups as we did for the `drivers` subsystems.

4.2.4 Observations on the *BSD kernels

The operating systems based on the BSD kernel constitute the most similar alternative to Linux-based operating systems in the libre software world. All BSDs derive from the UNIX version made at Berkeley since the 1970s. In particular, both FreeBSD and NetBSD are derivatives of the 4.4 BSDLite version released 1994, while OpenBSD is a branch (first released 1996) from NetBSD. These three BSD derivatives share architecture and a lot of code [Yamamoto *et al.*, 2005], as *copying* source code, or even entire files from other kernels is common practice [Fischer *et al.*, 2005].

As in the case of the Linux kernel, we have researched the growth of each of the BSD kernels as a whole and at the subsystem level. These kernels possess a set of characteristics that make the comparison with Linux worthy: they are from the same application domain (operating system kernels), are contemporary, are libre software and their source code base is similar in size (at least in order of magnitude).

System level growth for the *BSD kernels

Figure 4.10 displays the system growth for the BSD kernels starting in late 1995, when OpenBSD had its first commits. Even if all three kernels have achieved a significant size (2.5 MSLOC for NetBSD and over 1.5 MSLOC for OpenBSD and FreeBSD), we can see that their growth has not been super-linear.

NetBSD and FreeBSD show an almost linear growth pattern (see the values of the determination coefficient r^2 in table 4.6), which OpenBSD follows too, but only until 2001 (afterwards it loses large quantities of code in two occasions). We can identify a super-linear growth rate for FreeBSD until the

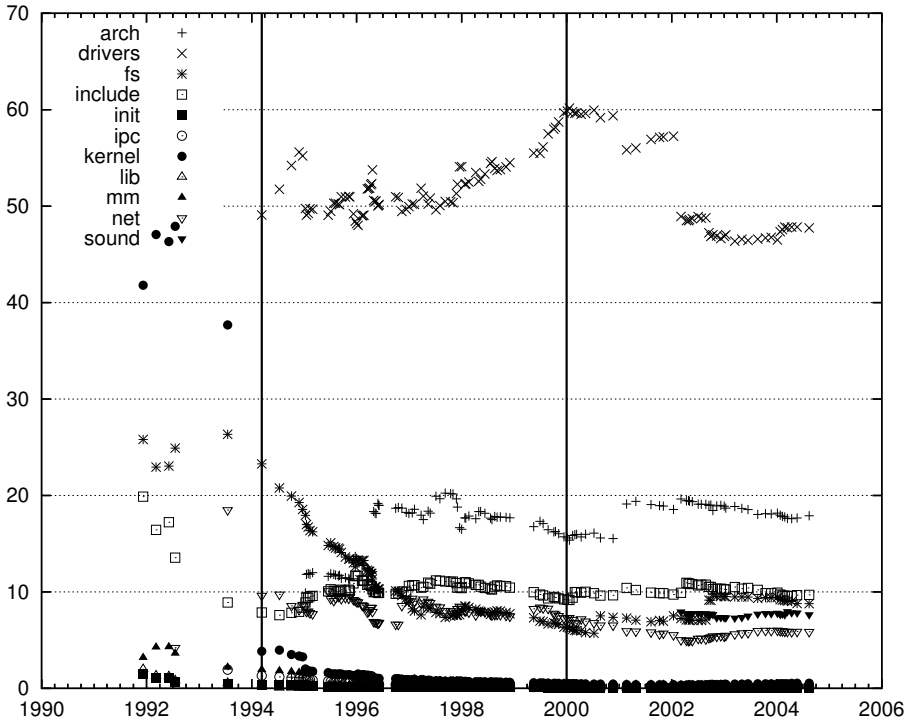


Figure 4.7: *Share of the subsystems of the Linux kernel over time - development releases only (vertical axis is given in percentages). A vertical line around 1994 gives the date of release for the version 1.0 of the Linux kernel.*

BSD kernel	Growth equation (linear fit)	r^2
NetBSD	$y = 610.2 \cdot t + 585731$	.993
FreeBSD	$y = 479.1 \cdot t + 200607$	.972
OpenBSD	$y = 240.2 \cdot t + 779607$	.891

Table 4.6: *Growth equation for the BSD kernels (based on statistical analysis)*

year 2000, which means that if Godfrey and Tu would have performed their study on FreeBSD, they would have found at the time of writing their paper similar patterns for Linux and FreeBSD. From 2000 onwards only Linux keeps on with such a super-linear growth, while FreeBSD seems to follow a more linear shape.

Growth at the subsystem level

As we have done for Linux, we study the subsystems of the three BSD kernels. Figure 4.11 shows the growth of the subsystems for FreeBSD kernel. The equivalent figures for the OpenBSD and NetBSD kernels can be found in figure 4.12. Subsystems do not grow super-linearly in any of the three cases, except for *dev* (devices), and only in its early stages. It is noteworthy that the shape of the *dev* subsystem is similar in all cases, possibly due to a common code base.

The *arch* subsystem is the largest one both in NetBSD and OpenBSD, although in the latter case it stops growing early in 2001 contributing a great deal to the shape of the total OpenBSD system.

Since one of the main goals of NetBSD is to work on as many platforms as possible, the larger size of *arch* is not a surprise, neither its continuous growth. OpenBSD supports many architectures, but with less drivers, and is the less growing system of the three, probably because of its strict security and auditing policies (the OpenBSD kernel is the base of an operating system targeted to environments with strict security constraints).

Summarizing our findings for the *BSD kernels: with the exception of FreeBSD and for some time before the year 2000, the most frequent growth pattern is linearity although there are some gaps at given moments that make the curves have erratic behaviours (and hence a low correlation coefficient).

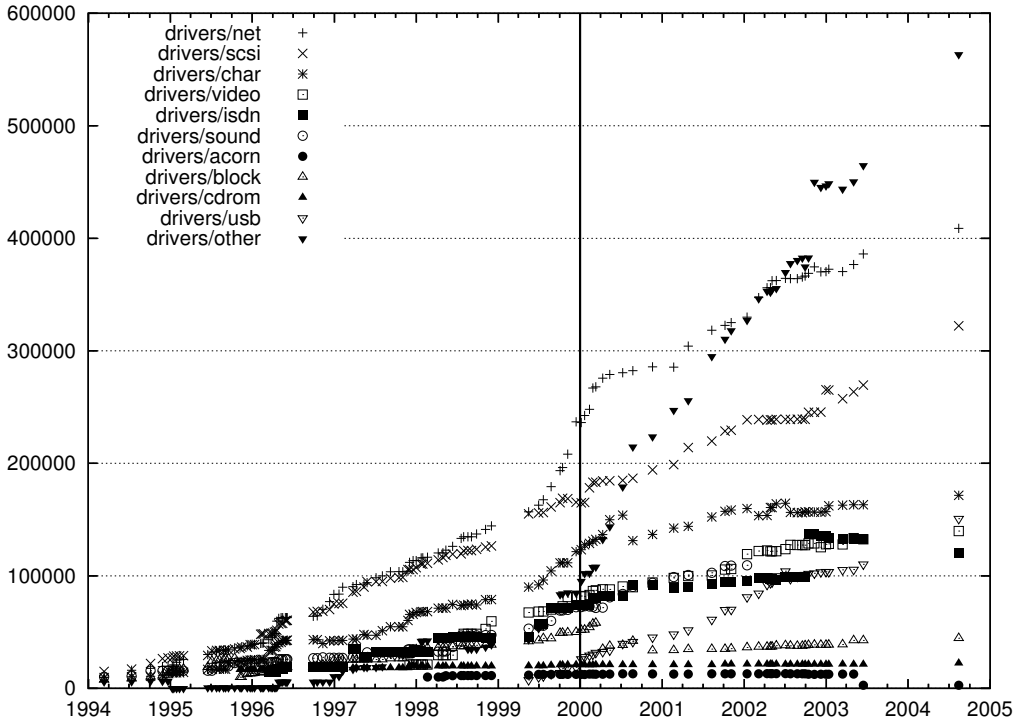


Figure 4.8: *Growth of the major drivers subsystems (development releases only); smaller drivers have been grouped together in* `drivers/other`.

The same is true at the subsystems level, where super-linearity is seldom achieved, and if at all not in a sustainable way as in Linux.

4.2.5 Observations on other libre software applications

We have studied 18 more large libre software systems to widen the sample, with the aim of applying the methodology to more cases, and to find out if results can be generalized. We have focused on projects which can be considered mature and with an active community of users and developers. In particular, we have selected projects with a large number of contributors (in the range of the hundreds or above), since a critical mass of developers has to be achieved to ensure the sustainability of the project.

Selected projects are usually considered typical libre software projects, although several different development models are found among them. However, all of them include voluntary and paid developers, external contributions, and interest in satisfying the needs of a sizable user community. The data for these systems has been obtained in April 2005, and at least four years of development are considered for all of them (details can be found in table 4.7).

In table 4.7 we include a summary of the evolution of the 18 libre software systems under study. For each row, the data about a system (its name can be found in the first column) is offered: the date when the system started to use a versioning system (notice this is different from the starting date of the project); the date for the 1.0 version (Ver 1.0), if available (in some cases that data is not available, such as for jakarta-commons, because it groups a set of subcomponents that are released independently, or in the case of Ant for which we have not found a 1.0 release, so we have inserted the date for the 1.1 release); in the *Prev* column whether a code base existed before starting the system's CVS (such as for GTK+ or The GIMP); whether strong ripples can be observed in the growth of the system (column *Rip*); the current size of the system (in lines of code); the linear growth function that fits the data for the system where t is given in months and y in lines of code; and finally the last column contains the correlation coefficient of the fit. The projects in the table have been ordered by their correlation coefficient, so that better linear fits have been put on top of the list while those projects with *bad* values appear at the bottom.

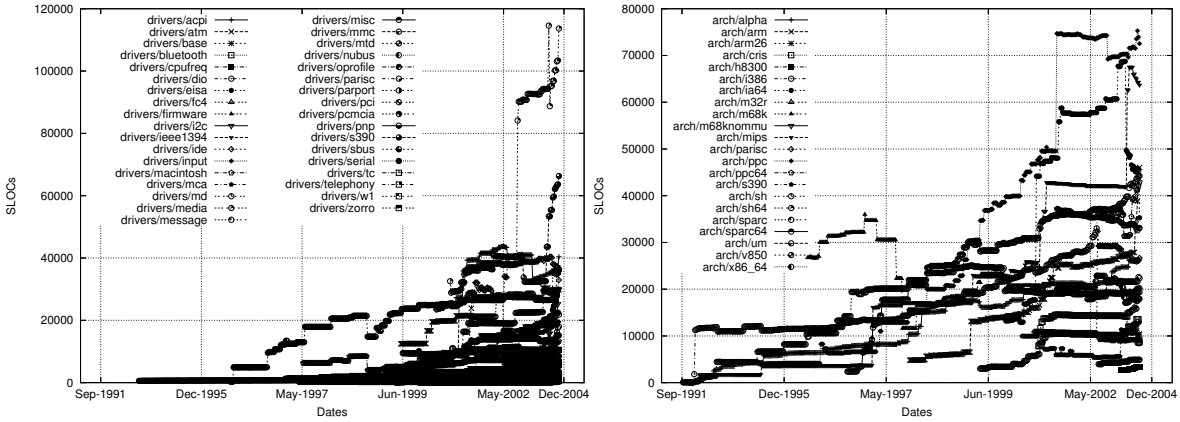


Figure 4.9: *Right: Growth of the* `drivers/others` *subsystems (development releases only). Left: Growth of the* `arch` *subsystem (development releases only). Vertical axis is given in SLOC.*

The reader can find a 3x6 matrix with growth plots for the 18 systems grouped in table 4.8. They are represented left to right and from top to bottom in the same order as they appear in table 4.7, so again projects that appear to have good linear fits appear at the top. The horizontal axis gives the time (in number of months since its inclusion in a CVS repository), while the vertical axis gives the software size in source lines of code. The straight line in the plots gives the function that has been obtained by fitting the data linearly.

Results for the other libre software applications

From the 18 projects that have been considered, at least 9 of them show a clear linear behavior throughout all the system life. We have fitted them to a linear function with r (correlation coefficient) values of about .99 and .98. However, all these systems have quite different sizes and start dates, which get reflected in different slopes of their growth lines.

It is also interesting to notice that for these projects it is not possible to infer from its growth plot when the version 1.0 was released as the pattern is the same before and after that release. In other words, from their growth plot it is impossible to determine the development (before version 1.0) and maintenance (after version 1.0) phases.

From the remaining 9 projects, 6 of them display also linear trends, although (strong or frequent) ripples in their growth curve cause them to be fitted to linear functions with coefficient values below .97 (but in all cases above .91). If those ripples are filtered out, many of them show behaviors which are similar to the ones observed in the first group.

Ripples are usually due to the inclusion of external code, the restructuring of the code base or the removal of code. All these actions have to be understood in a different manner in the libre software world than in *classical* software development environments. The reader should remember that the systems we are considering are part of larger projects, which means that code restructuring may happen not only intra-project (in the module itself), but also inter-project (to another module in the same repository). A large inclusion of external code is often possible legally because of a libre software license, and technically due to a modular structure of the code base. Removal of code may be a consequence of splitting the project in two. For instance, a sudden gap downwards may mean that some large part of the source code has been pulled out from a project to start a more specific one. This behaviour has been previously reported in the research literature on libre software: so, for instance, for the sake of modularity if the *core group* of developers grows larger than 15 to 20 developers, the project is split into smaller projects with the intention of improving manageability [Mockus *et al.*, 2002].

The remaining three projects, Ant, The GIMP and Evolution, have growth patterns that cannot be fitted to linearity (in fact, they show *bad* r values of .88, .87 and .80). While Ant yields a classical smooth growth (with some ripples early in the year 2002) as found in traditional software evolution

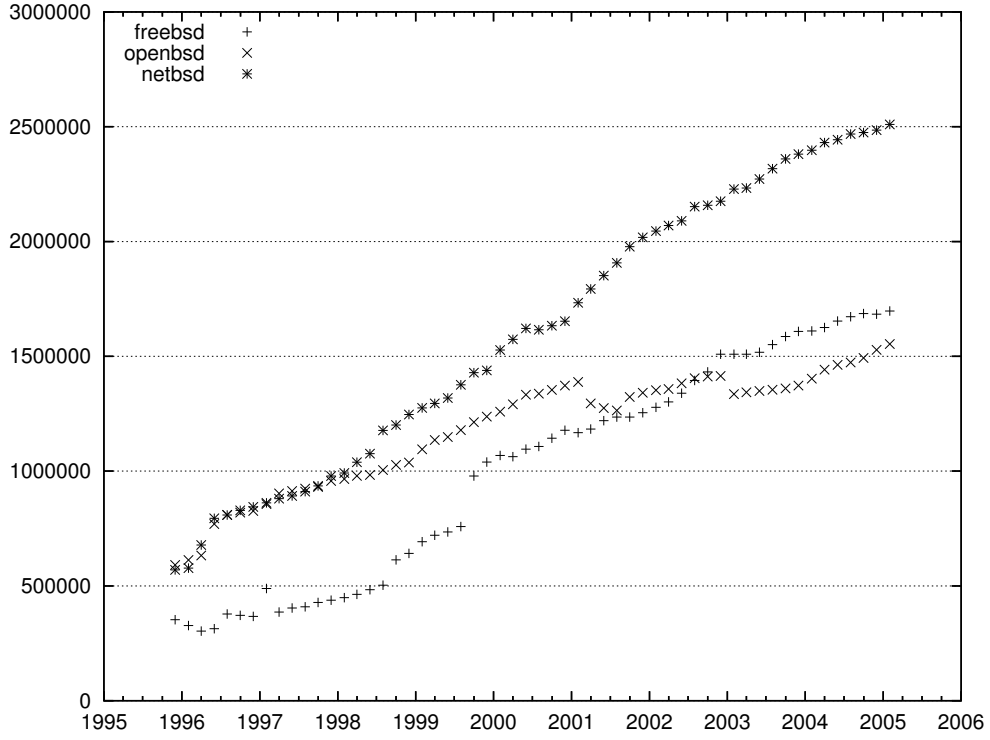


Figure 4.10: *Growth of the BSD derivatives*

Project	Start	Ver 1.0	Prev	Rip	Size	Growth Function	Correl. coef.
kdelibs	May 97	Jul 98	N	Y	615K	$y = 6421.1 \cdot t - 16474.8$	r = 0.995
jakarta-commons	Mar 01	-	N	N	429K	$y = 9394.7 \cdot t - 33888.0$	r = 0.994
mcs	Jun 01	Jun 04	N	N	1081K	$y = 26002.3 \cdot t - 105089.3$	r = 0.993
mono	Jun 01	Jun 04	N	N	222K	$y = 4912.9 \cdot t - 3436.6$	r = 0.992
koffice	Apr 98	Jan 01	N	N	780K	$y = 7965.3 \cdot t + 20724.8$	r = 0.992
kdepim	Jun 97	Jul 98	N	N	512K	$y = 4920.4 \cdot t - 32103.6$	r = 0.990
gnumeric	Jul 98	Jun 02	N	N	229K	$y = 3019.9 \cdot t + 17322.8$	r = 0.988
gtk+	Dec 97	Apr 98	Y	Y	388K	$y = 3371.7 \cdot t + 89968.9$	r = 0.985
xml-xerces	Nov 99	Oct 03	Y	Y	375K	$y = 4345.2 \cdot t + 104761.5$	r = 0.977
galeon	Jun 00	Dec 01	N	Y	90K	$y = 1460.0 \cdot t + 9095.4$	r = 0.967
httpd-2.0	Sep 99	Sep 02	Y	Y	127K	$y = 1000.5 \cdot t + 65668.8$	r = 0.947
xml-xalan	Nov 99	Oct 00	N	Y	337K	$y = 3896.0 \cdot t + 101817.1$	r = 0.943
kdebase	Apr 97	Feb 99	N	Y	362K	$y = 3097.1 \cdot t + 72062.8$	r = 0.935
kdenetwork	Jun 97	Jul 98	N	Y	293K	$y = 2142.9 \cdot t + 48781.0$	r = 0.933
kdevelop	Dec 98	Dec 99	N	Y	386K	$y = 4146.8 \cdot t - 22622.4$	r = 0.916
ant	Feb 00	(Aug 03)	N	Y	120K	$y = 1774.4 \cdot t + 15212.3$	r = 0.882
evolution	May 98	Dec 01	N	Y	208K	$y = 3801.2 \cdot t + 35801.7$	r = 0.842
gimp	Dec 97	Jun 98	Y	Y	557K	$y = 2696.7 \cdot t + 317718.9$	r = 0.815

Table 4.7: *Summary of the findings for a software evolution analysis applied to the projects listed in the first column. Start is the starting date of the CVS, Ver 1.0 the date of version 1.0 if available, Rip the appearance or not of ripples, size gives the size of the software in SLOC, the growth function is the linear fit and the correlation coefficient gives the quality of the fit.*

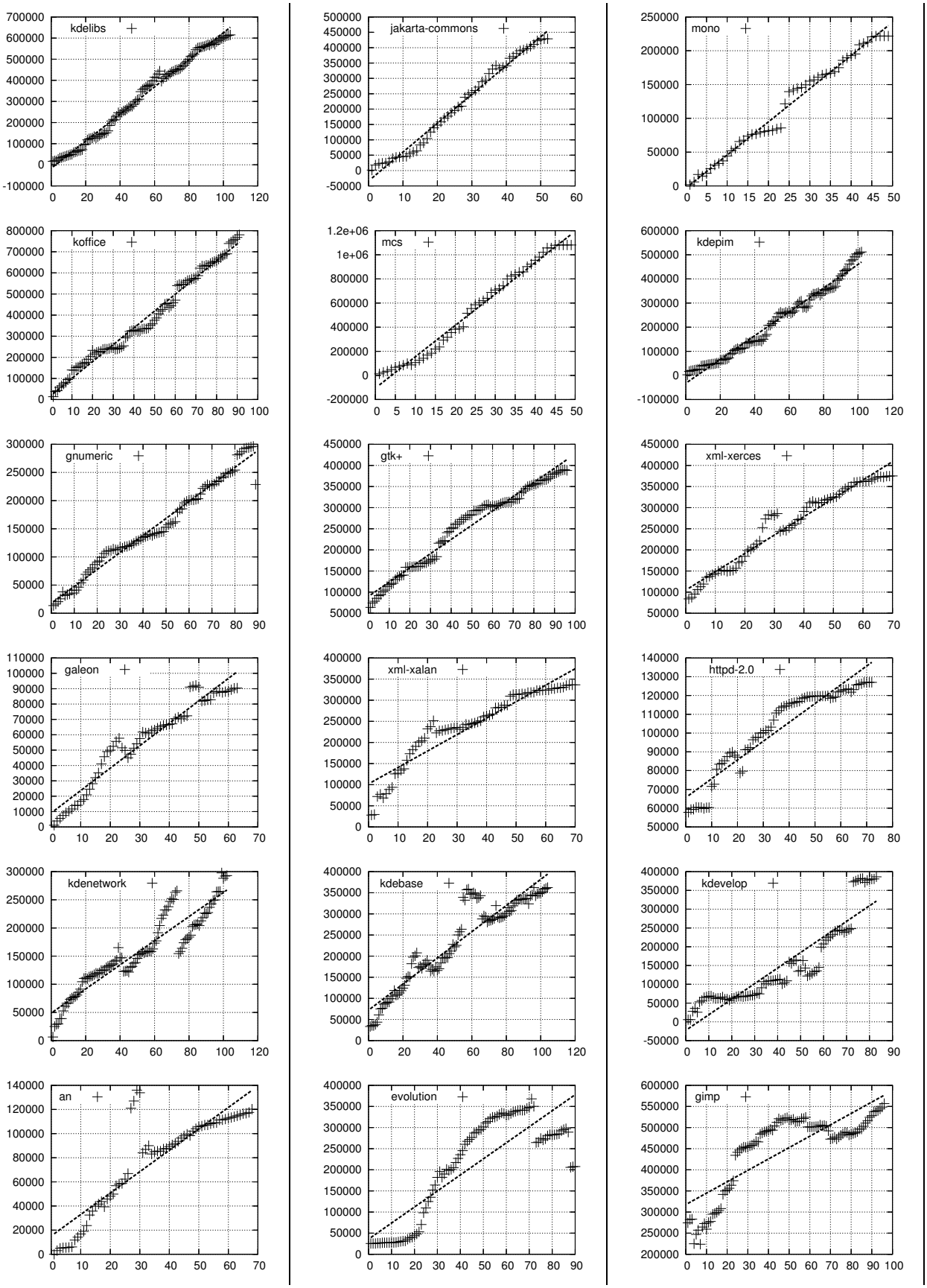


Table 4.8: *3x6 matrix with growth plots for 18 libre software systems. Projects with good linear fits have been situated at the top. The vertical axis is measured in SLOC; the horizontal axis in months since the project started to use a versioning system. More information can be found in table 4.7.*

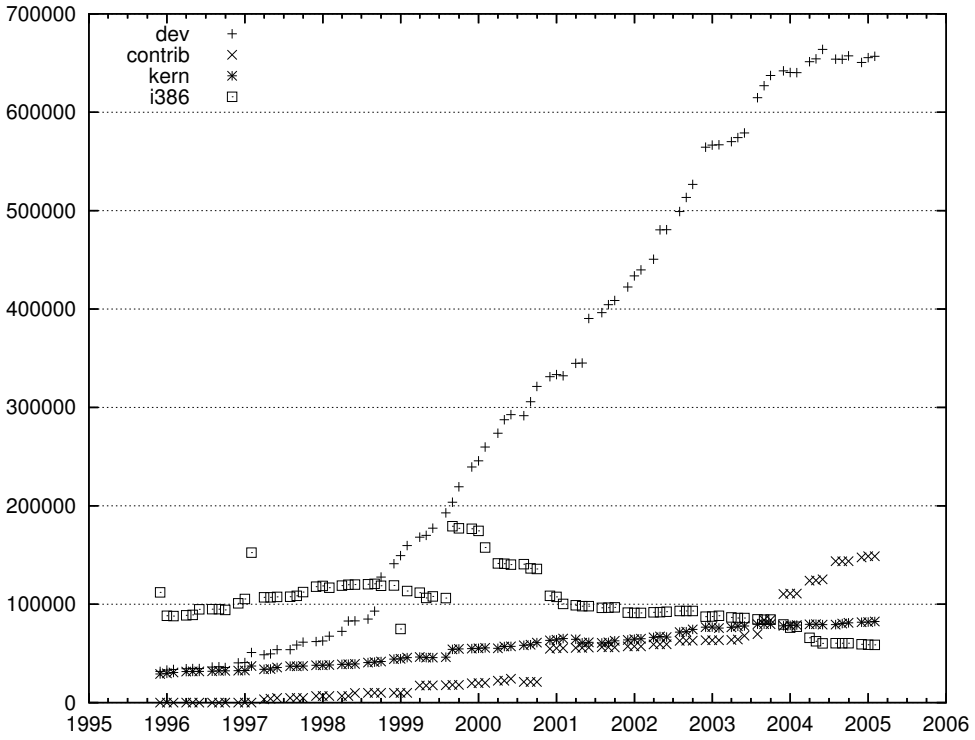


Figure 4.11: *Growth of the four largest subsystems of FreeBSD*

studies [Turski, 1996], the other two projects may be seen as exceptions or anomalies and hence require further explanation.

Evolution started as a small community-driven project, but about two years after its beginnings it was considered by a software company as a key software for its business model. The company hence hired some developers to work on it. This may explain the super-linear growth trend in its first stages, until version 1.0 was released. After that point, the growth follows the usual pattern identified by Turski, except for the heavy refactoring that has made the code base get smaller at least two times in the latest stages of development. Evolution is an exceptional case in the libre software world where large parts of the source code base are rewritten from scratch. Many developers see this not without polemic as it dismisses much debugging work on the old sources.

On the other hand, The GIMP was uploaded to the CVS only after three years of development, already with about 300,000 lines of code, which may cause some distortion. Then, until about 2000, we can observe an almost linear pattern. But since 2000 its growth has stagnated, in part because it has a mature architecture and most of the development around The GIMP is happening in modules outside what is considered The GIMP itself. A fast look at the GNOME CVS where The GIMP is stored results in 14 related modules with extra features and functionality.

Summarizing our analysis devoted to other applications which are not from the operating system kernels domain, we can state that 15 out of 18 systems follow a linear or close to linear growth pattern. Those that do not can be considered special in some sense.

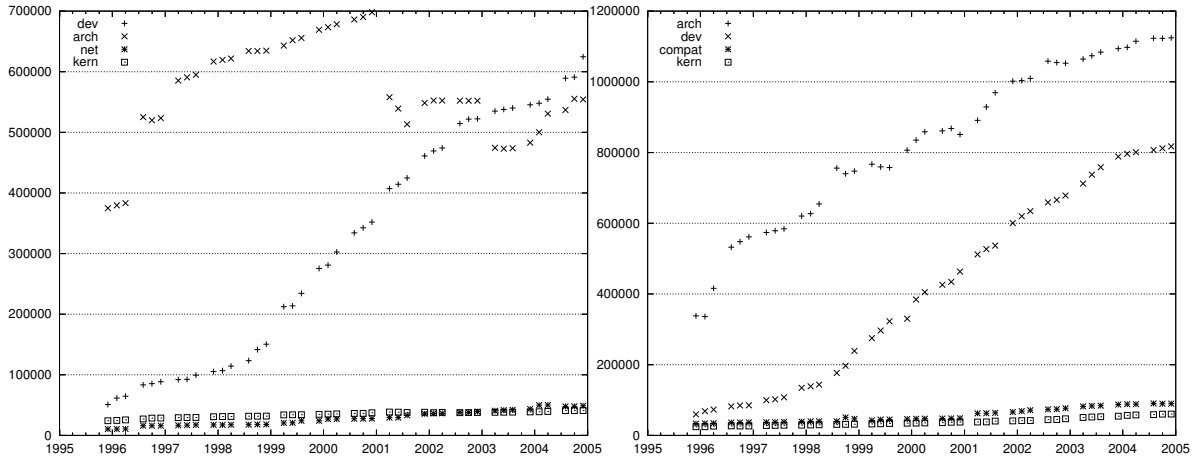


Figure 4.12: *Right: Growth of four most sized subsystems OpenBSD in the number of lines of code. Left: Growth of four most sized subsystems NetBSD in the number of lines of code.*

4.3 Evolution of software compilations

Software compilations, known generally as distributions, offer the possibility of installing, managing and updating software easily. Soon in the 1990s distributions started to gain momentum and today's popularity of the libre software movement is partly because of the fulfillment of those tasks.

There have been already some *radiographies* of some distributions, mainly of the well-known Red Hat and Debian distributions, in recent years pointing out what packages they contain, the size of the packages and the total distribution, some statistics on the number of files and on the programming languages, among other issues [Wheeler, 2001; González-Barahona *et al.*, 2001; Amor *et al.*, 2005b; 2005a].

The aim of our analysis in this thesis is to go a step beyond such type of *static* analyses and introduce the time axis in them. This can be considered as a clear influence from the software evolution practices. The evolution of software applications has been a matter of study for more than thirty years now [Lehman & Belady, 1985; Lehman & Ramil, 2001], but this type of studies have not been performed on software compilations. There are two main reasons for this: first, integrating independent software in a non-libre environment is both difficult technically (the availability of the source code is of great help) and legally (the license terms may avoid such an integration or at least require difficult agreements with the authors) and second, the possibility of studying those software compilation that have succeeded to surpass the aforementioned problems was limited (probably because of the same reasons).

4.3.1 Goals

The goals of this study differ slightly from the ones that could be considered as common for software evolution, in part because for the production of software compilations a different type of work has to be accomplished. Software compilations are based mainly on integration work rather than on development, although the latter, even if minoritary, is work that also has to be done (for instance, for the development of an installer or other software management tasks).

In any case, there are some aspects that are common to *traditional* software evolution analyses as how the size evolves. We will do this at the distribution level as well as for the components (packages) that compose the distribution - a sort of subsystem analysis as we have seen in the previous section. For packages, information about their size will also offer the possibility to see what type of packages are included and if larger ones or smaller ones are the predominant packages. A look at the largest packages in size will give us some further insight about the distribution. So, if the largest packages correspond to technical applications, the target users will most probably be IT-related users, while if many end-user solutions appear a wider audience may be under consideration.

But putting a software distribution together is not only integration work; maintenance has also to be performed. This has not to be understood in the classical way with corrective, adaptative or perfective maintenance activities as defined by Swanson [Swanson, 1976]. It may just be seen as integrating new versions of the software that have been released. In other words, a package maintainer may not submit any patches that correct errors; but he will have to update the package if new versions are published, integrating the new version. This raises interesting questions in our longitudinal analysis. For instance, we will analyze those packages that are kept and those that get *lost* over time, as the composition of the software compilation may vary. In addition, we will look at those packages whose version has not change (being probably a proxy for *unmaintained* packages).

As software compilations are composed of a large variety of software applications for different purposes and from different backgrounds, we may find a larger heterogeneity than when looking at a specific software applications. This is the case for instance in the use of programming languages; while a software application, as we have seen before in the case of Linux, is usually implemented primarily in a programming language being the rest of languages usually testimonial, this has not to be the case for software compilations.

On the other hand, studying compilations as large as the ones we have selected as case studies makes it possible to see them as a proxy of the whole libre software phenomenon. We are performing in this sense a holistic study of libre software and research how it is *in the large*, making it possible

to draw some conclusions about the phenomenon itself. In this sense, one of the analysis is an effort estimation analysis.

4.3.2 Methodology

The methodology that we have used for the analysis of the stable versions of Debian is as follows: first, we have retrieved the Sources.gz files which contain information about the packages that are distributed with every Debian distribution (for a detailed description of the Sources.gz, see 3.6.1). Then, each package is retrieved to a local machine, the number of source lines of code is counted and the programming language(s) in which the code is written is (are) recognized.

The counting is made by means of a tool called SLOCCount (presented in subsection 2.2.3). The results of the SLOCCount analysis are transformed afterward into an XML format that allows easy manipulation, visualization and transformations into other formats. Among the most interesting transformations we encounter the possibility of having the data in SQL format to store it in a database. With a simple web interface anyone can have access to raw data and more elaborated visualization forms that facilitate a first analysis (graphs, maps, among others). Many of the results carried out for this study are offered in a web site[3].

Distributions are organized internally in packages. Packages used to correspond to applications or libraries, although commonly Debian developers try to modularize packages to the maximum. So, for example, sources are frequently separated from documentation and data. This does not affect much our results, since the calculations that we have made consider the source lines of code solely, and those packages with documentation contain in general little or no code.

4.3.3 Observations on the size of Debian

We have selected the Debian GNU/Linux as the case study for this type of analysis. More information about Debian can be found in appendix A. At the moment of this thesis, the stable version of Debian has been just released some months ago and its version number is 3.1 (also known as *sarge*). The testing version has been codenamed *etch* and will become the next stable Debian version some time in the future. Finally, the one that is in development is *sid*. But in the past, *sarge* also passed through a testing phase and, before that, it was unstable/development. What we are going to consider in this thesis are the stable versions of Debian since version 2.0, published in 1998. Thus, we will see Debian 2.0 (alias *Hamm*), Debian 2.1 (*Slink*), Debian 2.2 (*Potato*), Debian 3.0 (*Woody*) and, finally, Debian 3.1 (*Sarge*).

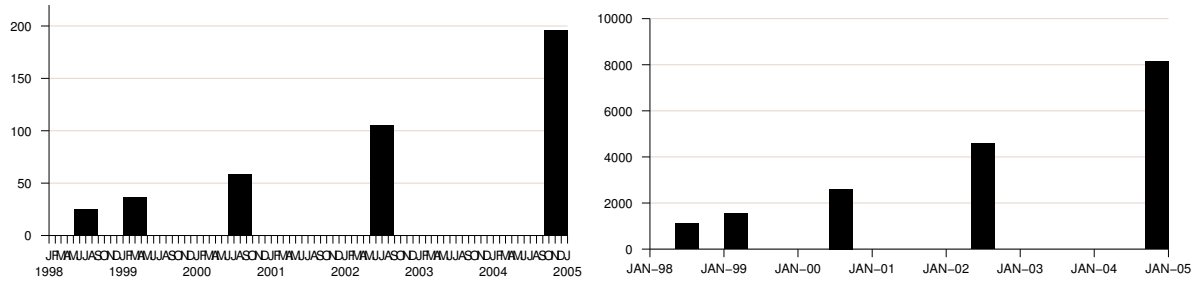


Figure 4.13: *Size, in MSLOC, and number of packages for the versions in study. Left: MSLOC for each version. Right: Number of packages for each version. Synopsis: In both graphics of this figure, the studied versions are spaced in time along the X axis according to their release date. On the left we can see the number of MSLOC that includes each version, while the right graph shows the evolution for the number of packages.*

In figure 4.13 the number of MSLOC and source packages for the considered stable versions of Debian can be found. Debian 2.0, released July 1998, included 1,096 source packages that had more than 25 MSLOC. The following stable version of Debian, the 2.1 (published around nine months later) contained more than 37 MSLOC in 1,551 source packages. Debian 2.2 (released 15 months after

[3]The web-site can be visited at following URL: http://libresoft.urjc.es/debian-counting.

Debian 2.1) summed up around 59 MSLOC in 2,611 packages, whereas the next stable version, Debian 3.0 (published two years after Debian 2.2), grouped 4,579 packages of source code with almost 105 MSLOC. Finally, almost three years later, Debian 3.1 has been released, with 8,141 source packages and more than 196 MSLOC.

Version	Release date	Source packages	Size (MSLOC)	Mean package size (SLOC)
Debian 2.0	July 1998	1,096	25	23,050
Debian 2.1	March 1999	1,551	37	23,910
Debian 2.2	August 2000	2,611	59	22,650
Debian 3.0	July 2002	4,579	105	22,860
Debian 3.1	June 2005	8,141	196	24,137

Table 4.9: *Size of the Debian distributions under study.*

Although the number of points is not sufficient to make an accurate model, we can infer from the current data that the Debian distribution doubles its size (in terms of source lines of code and of number of packages) around every two years, although this growth has been much more significant at the beginnings (from July 1998 to August 200 we had an increase of 135%) than in later releases (between July 2002 and June 2005 the source code base has not achieved a 100% increase even if 3 years have passed). Hence, using time in the horizontal axis, we would have a smooth growth of the software compilation as found by Turski [Turski, 1996]. On the other hand, if we considered only releases (which is the methodology preferred by Lehman), the growth would be super-linear basically because the time interval between subsequent releases has been growing for most recent releases.

4.3.4 Observations on the size of packages

In figures 4.14 and 4.15 we can see the graphs for the distribution of package sizes included in the versions of Debian. A small number of large packages in size (over 100 KSLOC) exist and the size of these packages tends to increase over time, as the sixth *law* of software evolution states [Lehman *et al.*, 1997]. Nevertheless, it seems surprising that in spite of the growth that Debian has undergone, the graph does not show big variations. Still more interesting is the fact that the mean size for the packages included in Debian is slightly regular (around 23,000 SLOC for Debian 2.0, 2.1, 2.2, 3.0 and 3.1, see table 4.9). With the data available at the present time it is difficult to give a forceful explanation of this fact, but we can suggest some thoughts. As packages tend to grow in size, if no new packages had been added to new versions of Debian, a growth in the mean package size would be expected. So it is the inclusion of new, small packages is what makes the mean size stay almost constant for around seven years. Perhaps the *ecosystem* in Debian is so rich that while many packages grow in size, smaller ones are included causing that the average to stay approximately constant.

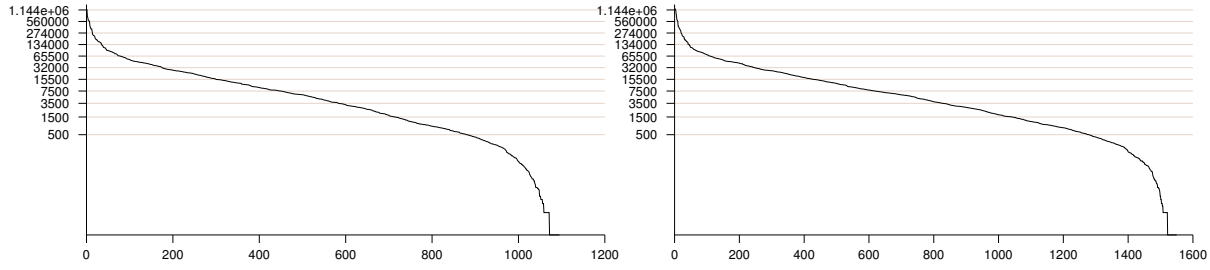


Figure 4.14: *Package sizes for Debian distributions. Packages are ordered by their size along the X axis, while the counts in SLOCs are represented along the Y axis (in logarithmic scale). Left: Debian 2.0. Right: Debian 2.1*

The histograms in figure 4.16 and figure 4.16 with package sizes displays the same data, but from another perspective. It can be clearly observed how large packages grow in size with time, while at the

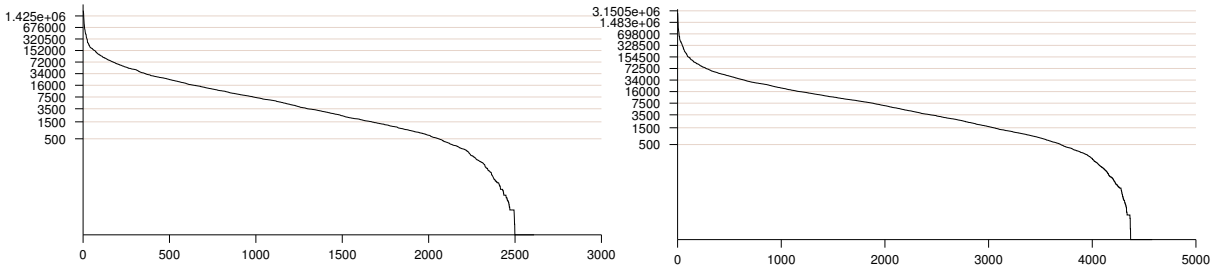


Figure 4.15: *Package sizes for Debian distributions. Packages are ordered by their size along the X axis, while the counts in SLOCs are represented along the Y axis (in logarithmic scale). Left: Debian 2.2. Right: Debian 3.0*

same time more packages near the origin appear, pointing out our previous findings. It is astonishing how many packages are *very small* packages (less than thousand lines of code), *small* (less than ten thousand lines) and *medium-sized* (between ten thousand and fifty thousand lines of code).

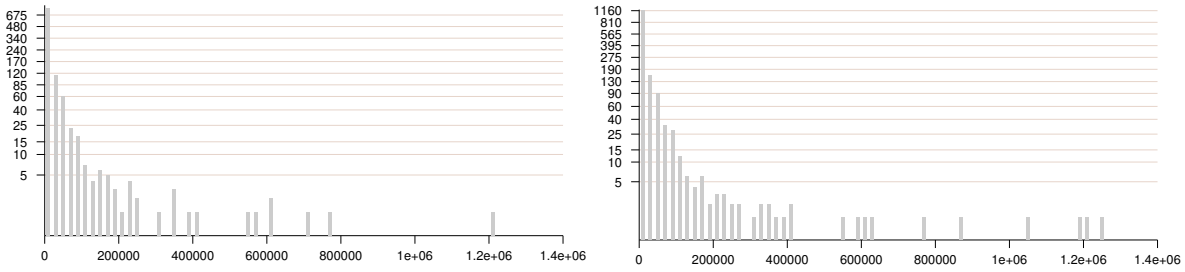


Figure 4.16: *Histogram with the SLOC distribution for Debian packages. Left: Debian 2.0. Right: Debian 2.1*

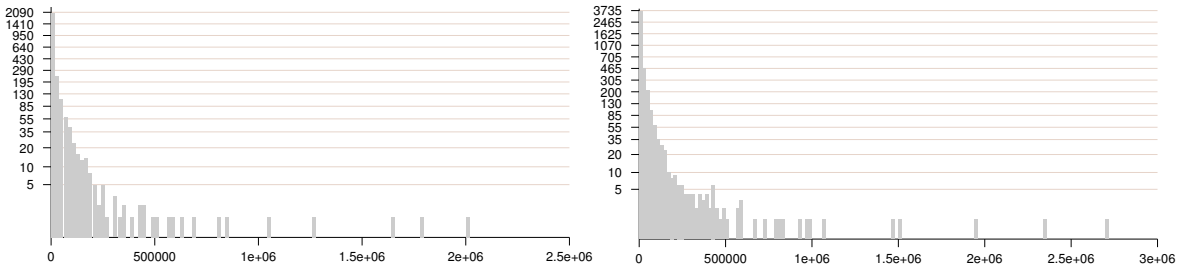


Figure 4.17: *Histogram with the SLOC distribution for Debian packages. Left: Debian 2.2. Right: Debian 3.0*

Largest packages

We have investigated the largest packages of each distribution. Many of these packages correspond to significant, well-known applications. From the study of these packages we can infer some information about the nature of the Debian distributions. Tables 4.10, 4.11, 4.12, 4.13 and 4.14 give the top ten list for the Debian versions that are considered.

There is much movement among the selected group. The fact that only the package which contains the Linux kernel prevails from the first considered version in this study, Debian 2.0, to the last one, Debian 3.1, after almost seven years is indicative in this sense. Regarding the domain of the

packages, we can see that while in earlier versions of Debian systems, software (xfree86, kernel-source) and developer (xemacs20, egcs, gnat, gdb, emacs20, lapack and gcc) tools were the most frequent applications in the list, in more recent versions there has been a shift towards end-user applications with the inclusion of the OpenOffice.org office suite and the Mozilla Firefox Internet browser. In some sense, this can be considered as an evidence that Debian has moved in these years from a mainly technical user audience to a wider audience, probably in accordance with the whole libre software movement.

	Package	Version	SLOC	files	SLOC/file
1.	xfree86	3.3.2.3	1,189,621	4,100	290.15
2.	xemacs20	20.4	777,350	1,794	433.31
3.	egcs	1.0.3a	705,802	4,437	159.07
4.	gnat	3.10p	599,311	1,939	309.08
5.	kernel-source	2.0.34	572,855	1,827	313.55
6.	gdb	4.17	569,865	1,845	308.87
7.	emacs20	20.2	557,285	1,061	525.25
8.	lapack	2.0.1	395,011	2,387	165.48
9.	binutils	2.9.1	392,538	1,105	355.24
10.	gcc	2.7.2.3	351,580	753	466.91

Table 4.10: *Top 10 packages in size for Debian 2.0.*

	Package	Version	SLOC	files	SLOC/file
1.	mozilla	M18	1,269,186	4,981	254.81
2.	xfree86	3.3.2.3a	1,196,989	4,153	288.22
3.	kernel-source	2.2.1	1,137,796	3,927	289.74
4.	prc-tools	0.5.0r	103,5230	3,025	342.22
5.	egcs	1.1.2	846,610	6,106	138.65
6.	xemacs20	20.4	777,976	1,796	433.17
7.	emacs20	20.5a	630,052	1,116	564.56
8.	gnat	3.10p	599,311	1,939	309.08
9.	gdb	4.17	582,834	1,862	313.02
10.	ncbi-tools6	6.0	554,949	951	583.54

Table 4.11: *Top 10 packages in size for Debian 2.1.*

	Package	Version	SLOC	files	SLOC/file
1.	mozilla	M18	1,940,167	9,315	208.28
2.	kernel-source	2.2.19.1	1,731,335	5,082	340.68
3.	pm3	1.1.13	1,649,480	10,260	160.77
4.	xfree86	3.3.6	1,256,423	4,351	288.77
5.	prc-tools	0.5.0r	1,035,125	3,023	342.42
6.	oskit	0.97.20000202	851,659	5,043	168.88
7.	gdb	4.18.19990928	797,735	2,428	328.56
8.	gnat	3.12p	678,700	2,036	333.35
9.	emacs20	20.7	63,0424	1,115	565.4
10.	ncbi-tools6	6.0.2	591,987	988	599.18

Table 4.12: *Top 10 packages in size for Debian 2.2.*

On the other hand, we can observe a clear tendency to an increase of the inferior limit of the top ten package list: whereas in Debian 2.0 GCC achieved the tenth position with 460 KSLOC, the last

	Package	Version	SLOC	files	SLOC/file
1.	kernel-source	2.4.18	2,574,266	8,527	301.9
2.	mozilla	1.0.0	2,362,285	11,095	212.91
3.	xfree86	4.1.0	1,927,810	6,493	296.91
4.	pm3	1.1.15	1,501,446	7,382	203.39
5.	mingw32	2.95.3.7	1,291,194	6,840	188.77
6.	bigloo	2.4b	1,064,509	1,320	806.45
7.	gdb	5.2.cvs20020401	986,101	2,767	356.38
8.	crash	3.3	969,036	2,740	353.66
9.	oskit	0.97.20020317	921,194	5,584	164.97
10.	ncbi-tools6	6.1.20011220a	830,659	1,178	705.14

Table 4.13: *Top 10 packages in size for Debian 3.0.*

Rank	Package	Version	SLOC	files	SLOC/file
1.	openoffice.org	1.1.2dfsg1	4,980,138	79,560	62.60
2.	kernel-source-2.6.8	2.6.8	4,023,484	16,075	250.29
3.	mozilla-firefox	0.9.3	2,476,383	24,932	99.33
4.	mozilla	2:1.7.3	2,411,922	33,556	71.88
5.	gcc-3.4	3.4.2	2,241,133	27,010	82,97
6.	insight	6.1+cvs.2004.08.11	1,690,058	9,658	174.99
7.	ace	5.4.2.1	1,391,794	27,952	49.79
8.	bigloo	2.6d+2.6e-alpha040622	1,325,772	2,084	636.17
9.	gdb-m68hc1x	1:6.2+2.92	1,274,279	5,360	237.74
10.	cernlib	2004.01.20	1,274,083	18,032	70.66

Table 4.14: *Top 10 packages in size for Debian 3.1.*

package in the top 10 for Debian 3.1, cernlib (a suite of data analysis tools and libraries created for experimentation in physics), consists of more than 1,274,000 lines of code. Hence, the amount of code required to be part of the *selected* group of largest packages in Debian has grown by almost 200% in the last seven years.

But top packages in size do not only tend to have more source code, they also show a tendency to have larger source code files. While the average in SLOC per file is in a rank between 352 and 359 for packages in the top ten, the average for all the packages in the same versions lies between 228 and 243 of source code lines per file. It exists, nevertheless, a big variance among the top packages. We have a range going from the 138 SLOC per file in version 1.1.2 of egcs (a derivative of the GNU GCC compiler) to the 806 SLOC per file in bigloo (a system for Scheme compilation) in its version 2.4b.

4.3.5 Observations on the maintenance of packages

Up to the moment, we have seen how Debian has been growing in the last 7 years as far as the number of packages and the number of SLOC is concerned. In the following paragraphs, we will attend the opposite dimension: packages that have not changed. This has to be understood in the sense that taking care of a software distribution requires maintaining packages, i.e. among other activities including new versions of the packages in the distribution. Packages that maintain from one release to the other the same version number may have been maintained actively, but usually even performing corrective maintenance implies releasing new versions of the software. So, we can assume that if the version number has not been changed then changes have been minor and do not need to be considered.

Figure 4.3.5 will help explaining how we are going to measure maintenance activity supposing that we have two distributions (given each one by a set of packages, in the figure these are Debian 2.0 and Debian 3.1). The circle that gives the set of packages for the Debian 3.1 version has a larger radius as it contains many more packages than Debian 2.0 (the area of the circles could be considered

as proportional to their size in number of packages). Both sets may have packages in common (the intersection between the two sets, as it is the case for the kernel-source package). Other packages will only be included in one of them. If packages appear only in the older Debian version, we say that it has been *lost*, while packages that appear only in the newer one are new (or added) packages. We can also identify a subset of those packages that have remained with the same version number (a subset of the intersection between the two sets); those are the packages that we will consider unmaintained.

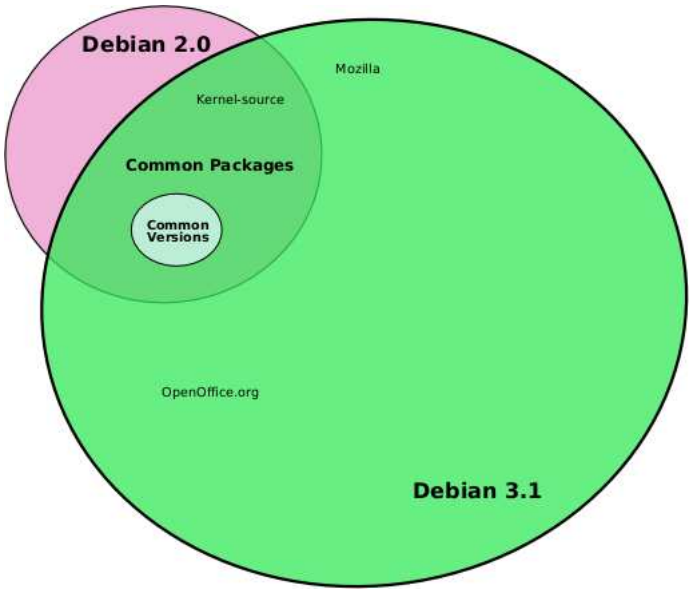


Figure 4.18: *Illustration of common packages between Debian 2.0 and Debian 3.1. Among these packages, we may find a subset that has the same version number.*

Tables 4.15, 4.16, 4.17, 4.18 and 4.19 contain some statistics about common packages in different stable versions. As explained before, we assume that two versions have a package in common, if that package is included in both, independently of the version number of the package. Each table displays in its second column the number of packages in common that a version of Debian has with the other versions. To facilitate the comparison in relative and absolute terms, the same version of Debian that is compared is included. Needless to say, Debian 2.0 will have in common with itself 1,096 (all) source packages.

Out of the 1096 packages included in Debian 2.0 only about 800 appear in the last version of Debian considered in this thesis. This means that around 25% of the packages have disappeared from Debian in seven years. The number of packages of the 3.0 version that are still included in 3.1 is 3,848 out of 4,578 which gives us a similar percentage of *lost* packages.

On the other hand, we have to consider that distributions contain applications and libraries that evolve. This can be observed from the fact that the own version number of packages evolve. For example, the Linux sources are included generally in a package called kernel-source. In each version of Debian, the version number of kernel-source changes, so we can state that Linux has evolved and that these changes and improvements have been introduced in Debian. But this does not have to be the case for all packages. In the same way that we were previously interested in packages in common without mattering if the version numbers had changed, now we are going to consider those packages with version numbers that have not varied. Hence, we will identify packages included in two different Debian versions that have the same package version. Again, we add the own Debian version being compared. Because of that Debian 2.0 will have all of its packages (1,096) in common with itself.

The fact that Debian 3.1 includes 158 packages that have not evolved since Debian 2.0 is very surprising, as 15% of the source packages included in Debian 2.0 have stayed almost with no alterations since they were introduced seven years ago (or earlier). As expected, the number of packages with versions in common increases for neighbouring distributions.

Version	Com pkgs	Com vers.	SLOC com vers.	Files com vers.	SLOC com pkgs
Debian 2.0	1,096	1,096	25,267,766	110,587	25,267,766
Debian 2.1	1,066	666	11,518,285	11,5126	26,515,690
Debian 2.2	973	367	3,538,329	86,810	19,388,048
Debian 3.0	754	221	1,863,799	70,326	15,888,347
Debian 3.1	813	158	1,271,377	15,296	15,594,976

Table 4.15: *Packages and versions in common for Debian 2.0*

Version	Com pkgs	Com vers.	SLOC com vers.	Files com vers.	SLOC com pkgs
Debian 2.0	1,066	666	11,518,285	115,126	26,515,690
Debian 2.1	1,551	1,551	37,086,828	161,303	37,086,828
Debian 2.2	1,384	602	8,460,239	133,140	30,052,890
Debian 3.0	1,076	322	3,152,790	108,071	24,743,063
Debian 3.1	1,124	231	2,306,969	27,543	23,630,211

Table 4.16: *Packages and versions in common for Debian 2.1*

Version	Com pkgs	Com vers.	SLOC com vers.	Files com vers.	SLOC com pkgs
Debian 2.0	973	367	3,538,329	86,810	19,388,048
Debian 2.1	1,384	602	8,460,239	133,140	30,052,890
Debian 2.2	2,610	2,610	59,138,348	257,724	59,138,348
Debian 3.0	1,921	771	8,356,302	186,508	42,938,562
Debian 3.1	1,946	508	4,992,308	60,525	36,584,110

Table 4.17: *Packages and versions in common for Debian 2.2*

Version	Com pkgs	Com vers.	SLOC com vers.	Files com vers.	SLOC com pkgs
Debian 2.0	754	221	1,863,799	70,326	15,888,347
Debian 2.1	1,076	322	3,152,790	108,071	24,743,063
Debian 2.2	1,921	771	8,356,302	186,508	42,938,562
Debian 3.0	4,578	4,578	104,305,557	403,285	104,702,397
Debian 3.1	3,848	1,567	16,042,810	211,299	78,451,818

Table 4.18: *Packages and versions in common for Debian 3.0*

Version	Com pkgs	Com vers.	SLOC com vers.	Files com vers.	SLOC com pkgs
Debian 2.0	813	158	1,271,377	15,296	15,594,976
Debian 2.1	1,124	231	2,306,969	27,543	23,630,211
Debian 2.2	1,946	508	4,992,308	60,525	36,584,110
Debian 3.0	3,848	1,567	16,042,810	211,299	78,451,818
Debian 3.1	8,134	8,134	196,499,111	2,355,220	196,499,111

Table 4.19: *Packages and versions in common for Debian 3.1*

4.3.6 Observations on the programming languages

Our methodology implies to identify the programming language of source code files before counting the number of SLOCs. Thanks to this, we are able to compute the significance of the different programming languages in Debian. The most used language in all Debian versions is C with percentages that vary between 55% and 85% and with a big advantage on his immediate pursuer, C++. It can be observed, nevertheless, that the importance of C is diminishing gradually, whereas other programming languages

are growing at a good rate.

For example, in table 4.20 the evolution of the most significant languages - those that surpass 1% of code in Debian 3.1 - is shown. Below the 1% mark we can find, in this order, tcl, Ada, PHP, Pascal, ML, ObjC, YACC, Csharp, Lex, Awk, Sed and Modula3.

	2.0	% 2.0	2.1	% 2.1	2.2	% 2.2	3.0	% 3.0	3.1	% 3.1
C	19,371	76.7%	27,773	74.9%	40,878	69.1%	66.6	63.1%	106.4	54.1%
C++	1,557	6.2%	2,809	7.6%	5,978	10.1%	13.1	12.4%	35.8	18.2%
Shell	645	2.6%	1,151	3.1%	2,712	4.6%	8.6	8.2%	18.4	9.4%
Perl	425	1.7%	774	2.1%	1,395	2.4%	3.2	3.0%	6.8	3.4%
Lisp	1,425	5.6%	1,892	5.1%	3,197	5.4%	4.1	3.9%	6.8	3.4%
Python	122	0.5%	211	0.6%	349	0.6%	1.5	1.4%	3.6	1.8%
Java	22	0.1%	58	0.2%	183	0.3%	0.5	0.5%	2.8	1.4%
Fortran	494	2.0%	735	2.0%	1,182	2.0%	1,939	1.8%	2.7	1.4%

Table 4.20: *Top programming languages in Debian. For Debian 2.0, 2.1 and 2.2 the sizes are given in KSLOC, for versions 3.0 and 3.1 in MSLOC.*

There exist some programming languages that we could consider as minor languages and that reach a quite high position in the classification. This is because although being present in a reduced number of packages, these are quite large in size. That is the case of Ada, that sums up 430 KSLOC in three packages (gnat, an Ada compiler, libgtkada, a binding to the GTK library, and Asis, a system to manage sources in Ada) of a total of 576 KSLOC that have been identified as code written in Ada in Debian 3.0. A similar case is the one for Lisp, that counts with more than 1.2 MSLOC only for GNU Emacs and XEmacs of around 4 MSLOC in the whole distribution.

The programming language distribution pie-charts display a clear tendency in the decline of C. Something similar seems to happen to Lisp, that was the third most used language in Debian 2.0 and has become the fifth in Debian 3.1 (in fact, in 3.1, the forth language is Perl), and that foreseeably will continue backing down in the future. In contrast, the part of the pie corresponding to C++, shell and other programming languages grows.

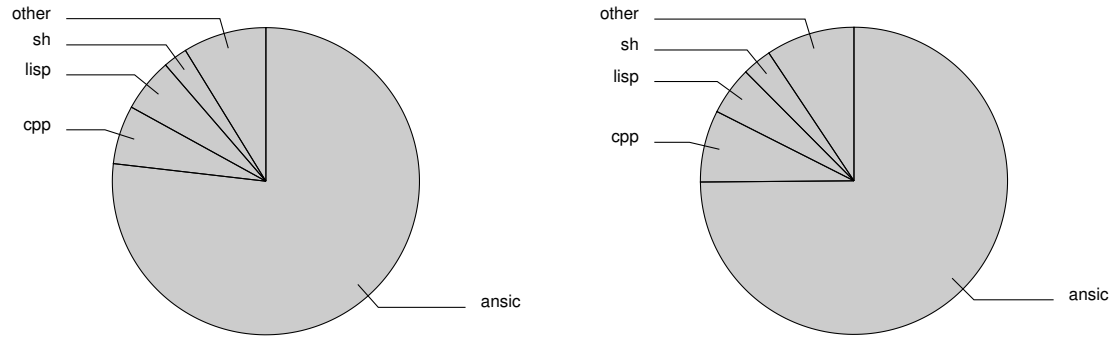


Figure 4.19: *Pie with the distribution of source lines of code for the predominant languages in Debian. Left: Debian 2.0. Right: Debian 2.1*

Figure 4.23 provides the relative evolution of programming languages which gives a new perspective of the growth for the last five stable Debian versions. We take therefore the Debian 2.0 version as reference and suppose that the presence of each language in it is 100% (normalized to 1) so that growth for a programming language is shown relative to itself. The graph should be read as follows: for each line in Debian 2.0 for a given language, the figure gives the number of lines in subsequent Debian releases for that language.

Previous pies evidenced that C is backing down as far as its relative importance is concerned. In this one we can observe that C has grown more than 300% throughout the four versions. But we can

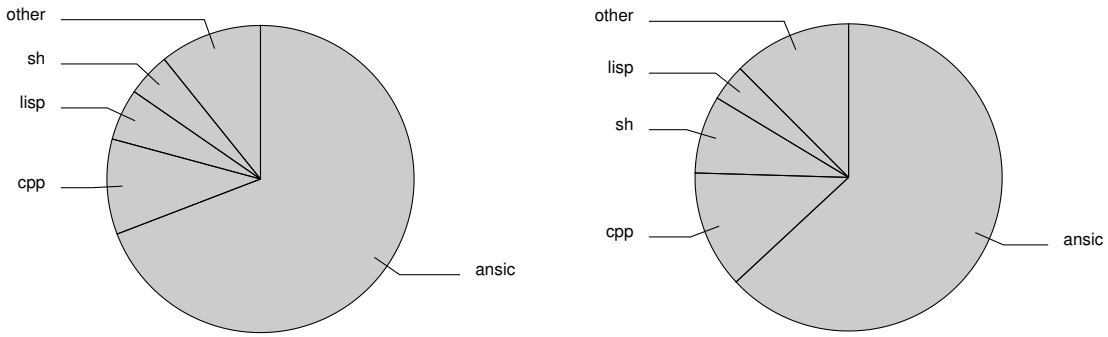


Figure 4.20: *Pie with the distribution of source lines of code for the predominant languages in Debian. Left: Debian 2.2. Right: Debian 3.0*

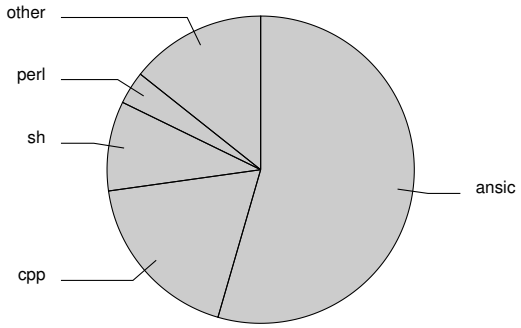


Figure 4.21: *Pie with the distribution of source lines of code for the predominant languages in Debian 3.1*

see that scripting languages (shell, Python and Perl) have undergone an extraordinary growth, all of them multiplying their presence by factors superior to seven, accompanied by C++. Languages that grow a smaller quantity are the *traditional*, compiled ones (Fortran and Ada) and others (such as Lisp, a *traditional* language that does not require compilation). This can give an idea of the importance that interpreted languages have begun to have in the libre software world.

Figure 4.23 includes the most representative languages in Debian, but excludes Java and PHP, since the growth of these two has been enormous, in part because their presence in Debian 2.0 was testimonial, in part because their popularity in the latest time is beyond doubt.

4.3.7 Observations on the size of files

It should be remarked that some of the most important programming languages have spectacular increases in their use, but that their mean file sizes remain generally constant (see table 4.21). Thus, for C the average length lies around 260 to 280 SLOC per file, whereas in C++ this value is located in an interval going from 140 to 185. We can find the exception to this rule in the shell language, that triples its mean size. This may be because the shell language is very singular: almost all the packages include something in shell for their installation, configuration or as *glue*. It is probable that this type of scripts get more complex and thus grow over the years.

It is very peculiar to see how structured languages usually have larger average file lengths than object-oriented languages. Thus the files in C (or Yacc) usually have higher sizes, in average, than those in C++. This makes us think that modularity of programming languages is reflected in the mean file size.

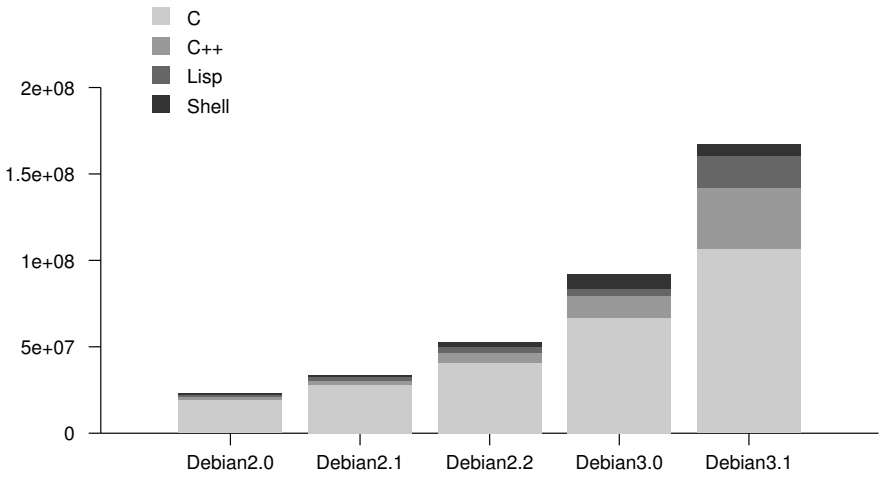


Figure 4.22: *Evolution of the four most used languages in Debian*

Language	Debian 2.0	Debian 2.1	Debian 2.2	Debian 3.0
C	262,88	268,42	268,64	283,33
C++	142,5	158,62	169,22	184,22
Lisp	394,82	393,99	394,19	383,60
shell	98,65	116,06	163,66	288,75
Yacc	789,43	743,79	762,24	619,30
Mean	228,49	229,92	229,46	243,35

Table 4.21: *Mean file size for some programming languages.*

4.3.8 Effort and time estimation

The COCOMO model [Boehm, 1981], introduced with some detail in subsection 2.2.2, gives an estimation of the human and monetary effort that is necessary to generate software for a given size. Although this model is not appropriately suited for libre software development, the results may give us an idea of the order of magnitude of the costs that creating Debian would represent, giving us the necessary optimal efforts, if a proprietary, waterfall development model had been used.

In general, the most astonishing result that COCOMO offers is its cost estimation (see table 4.22). Some words should be said in order to clarify the concept. In this estimation two factors are considered: the average developer salary and the factor of *overhead*. In the calculation of cost estimation, the average wage for a full-time system programmer has been taken from the year 2000 salary survey[4]. *Overhead* is the cost that any company has to assume independently from the programmers' salaries. The cost of having secretaries, a marketing team and other non-technical staff has to be added to the costs of photocopies, electricity, equipment or hardware, among others. For our calculations, we have supposed an overhead factor of 2.4. In summary, the final cost calculated by COCOMO is the total cost that a company would have to confront to create a software of the specified size and not simply the money that programmers would perceive to create the software. Once this is understood, cost calculations seem less bulky.

In table 4.22 we can observe the results of applying the basic COCOMO model to the various Debian stable versions. The results have been obtained by means of the separate calculation of the cost that each package would suppose. The amounts for each package have been summed up to give the total cost. It should be noted that as COCOMO is a non-linear model, the sum of the separate costs of packages is not equal to the cost of considering everything as one single project. The first

[4]This data is originally from the Computer World 2000 Salary Survey. The original report can be found at http://www.computerworld.com/cwi/careers/surveysandreports

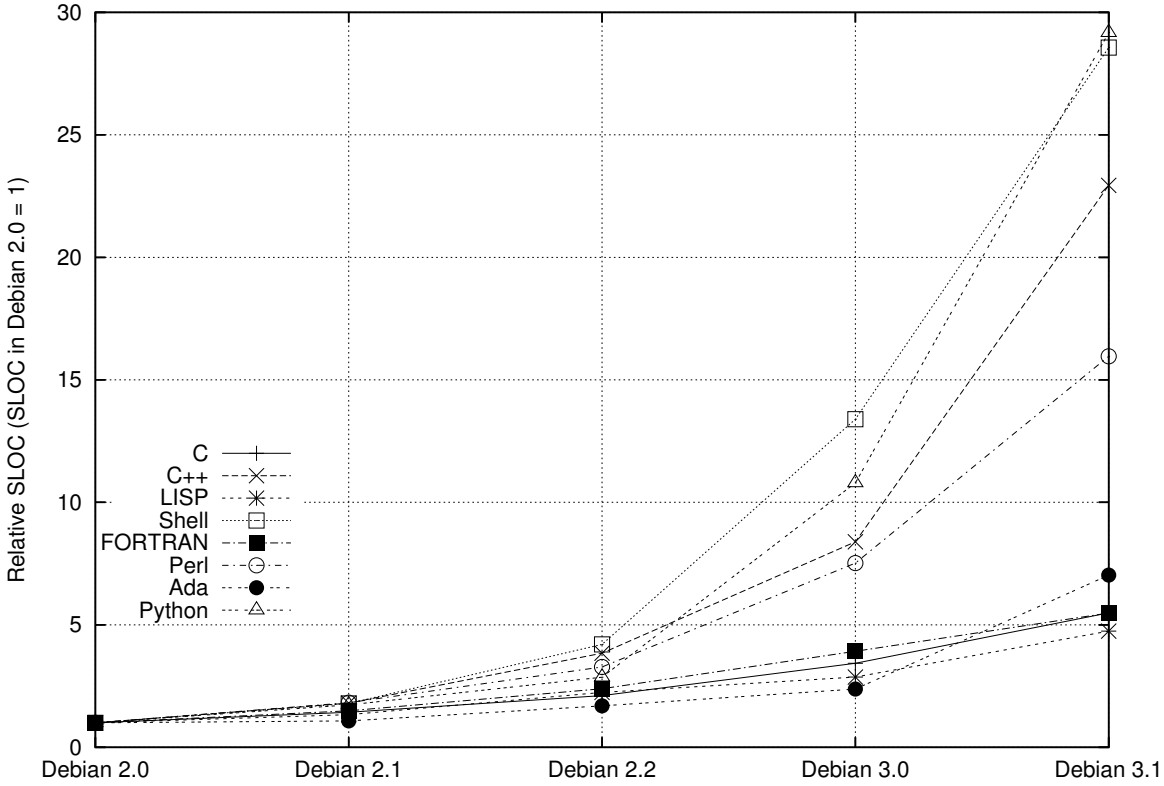


Figure 4.23: *Relative growth of some programming languages in Debian*

result yields the inferior effort limit, since integration tasks are not considered, whereas in the second case we would obtain the superior limit, since savings from having independent projects would have not been taken into account. As stated before, for our current goals it is enough with an estimation of the order of magnitude and therefore results for only one cost model have been calculated.

Version	MSLOC	Effort (man-years)	Time (years)	Cost (USD)
Debian 2.0	25	6,360	4.93	860,000,000
Debian 2.1	37	9,425	4.99	1,275,000,000
Debian 2.2	59	14,950	6.04	2,020,000,000
Debian 3.0	105	26,835	6.81	3,625,000,000
Debian 3.1	196	50,482	8,68	6,819,000,000

Table 4.22: *Effort, time and development cost estimation for each Debian version.*

4.4 Software Archaeology

Software evolution is based on considering the study of a software system by obtaining some snapshots over time (being the snapshots releases in the *classical* methodology, although they just could be periodically-spaced as we have used them in our analysis in section 4.2). As depicted in figure 4.24, for the software evolution metaphor the software engineer is aware of how the software has changed. Based upon this observations, he may derive some findings.

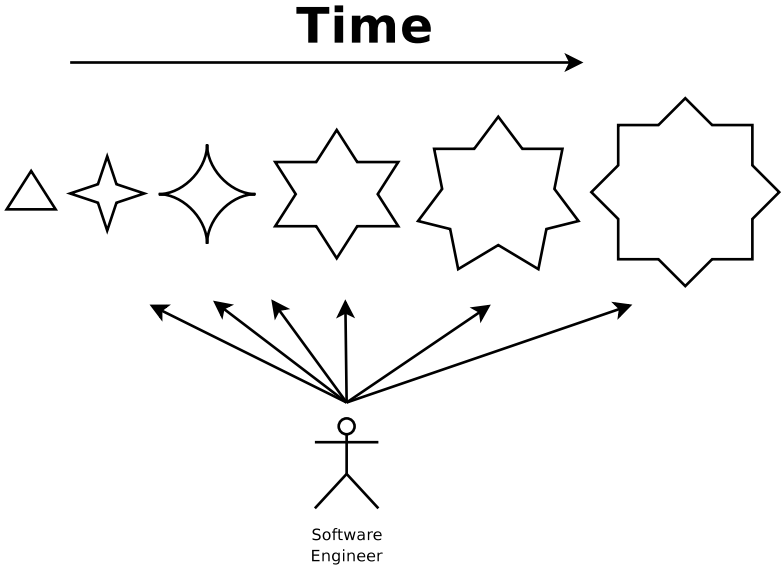


Figure 4.24: *Software evolution point of view. The software engineer views how the software system changes.*

In general, the view that is provided by software evolution implies a situation that is not that important when the maintenance of a software system is intended. In these cases, it is the current state of the sources which becomes the most important issue, being all previous states less important. Figure 4.25 shows the idea behind the approximation of software archaeology; the software engineer looks backwards *through* the current state of the software which is the one he will have to lead with. The current state of the software will be heavily influenced by previous states of the software system, but only the ones that have persisted are of interest, i.e. code and other artifacts that cannot be found in the latest version are uninteresting.

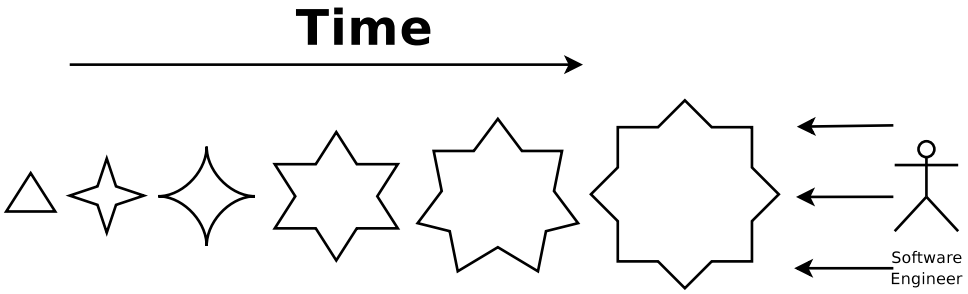


Figure 4.25: *Software archaeology point of view. The software engineer views from the current state of the software into the past.*

Hence, in this section we present a new approach for software maintenance based on this new view. A central concept of this idea is the *age* of the software components that are going to be maintained, being the software components at the line-of-code level. So, extracting information of when a line was last modified and by whom is the basis of this approach. Doing this for all lines in the software system will provide us with a rich data set that will help managing the software maintenance process. On top of this, we will define a set of indexes that will help in the application of software archaeology analyses both for knowing the situation of a project and for comparing it with others. Following the empirical

nature of this thesis we will apply our idea on some libre software projects and discuss limitations and future lines of research.

4.4.1 Goals

The primary goal of this section is to find out if a methodology based on software archaeology may be seen as a proxy to measure the software maintenance effort in (libre) software projects. In addition, we are interested in finding if our approach may help in determining the maintainability of projects in the near future. The key ideas behind such analysis are:

- A piece of software is easier to maintain if it has been recently been worked on.

  This principle is based on the experience that every developer has that maintaining a piece of code written by himself recently is, in general, easier than maintaining some code that was written some months ago. There are some factors that contribute to soften this situation, as in-line documentation and a good code structure, but in general and for the same code this can be considered as a valid assumption.

- Maintaining software created by a third person is more difficult than an own piece of software.

  Although again this can be mitigated by writing *better* code (structure, documentation, among others), a code written by a developer will be more easy to understand and maintain that code written by someone else. The corollary for this is that maintaining the same code requires less effort to the author of that code than to any other person.

- Having the possibility of asking the original author of the software improves maintenance.

  This is a consequence from the two previous points. If a developer has to maintain a piece of software written by somebody else, if the original author is still part of the development team, we can count on his knowledge. If the original author has abandoned the project, he has *taken* his knowledge with him.

Regarding the first idea, we will have to research how many lines remain from the past remain and when these lines were added. This will give us an idea of how much the system has been maintained so far, but also of how difficult it will be to maintain the software in the future; code that is older will be less maintainable. The second idea will introduce those developers who have authored the lines and provide information about how many of those developers are still part of the team. As we have enunciated in the third idea, having those developers still in our development team will raise maintainability, even if they are not directly in care of the maintenance.

As case studies, we have selected nine libre (free, open source) software projects that are, with some exception, among the hundred largest libre software applications included in the latest stable Debian GNU/Linux release[5].

4.4.2 Methodology and case studies

The methodology for the software archaeological analysis has been presented in subsection 3.3.3. We have selected a set of case studies from the libre software world in order to apply our methodology to real-life software systems.

We have introduced a procedure that does not consider blank lines and comments. In order to verify that our filtering procedure is accurate enough, we compute the source lines of code with SLOCCount[6] (see subsection 2.2.3). The counts given by SLOCCount have been compared with the number of lines obtained with our procedure. Table 4.23 shows that the error rate is low enough to consider our approach sufficient. The reason why SLOCCount always detects more lines of code is

[5]Facts and figures about the largest libre software applications included in the latest stable Debian release can be found at: http://libresoft.urjc.es/debian-counting/sarge.

[6]We use the '–duplicates' option which counts duplicated files twice as our tools in opposition to SLOCCount does not filter them out. In any case, this supposes less than 0.5% for the projects under consideration

because its heuristics look at the content of the files in addition of the extension (so more files are identified).

Regarding branches, we have merged them with the next major version number. This has been the case for a very small amount of lines of the Mozilla source code base: we found revision numbers 1.1.2 and 1.1.3 that were merged to 1.2. This supposed changing 37 lines affecting 9 files of the almost 10 million lines of code.

The projects under study have been chosen so that they differ in significant elements (age, size in SLOC, complexity, number of developers, among others), being the only common characteristic their libre nature and the fact that they are currently distributed with all major Linux distributions (which is an evidence of their popularity). A short description of the applications can be found in appendix A.

In total, our nine case studies sum up 9.5 millions lines of code mainly written in C and C++. The number of analyzed source code files is 52,975. Table 4.23 presents the most important facts about our case studies, while the next lines are devoted to the reasons why we have chosen them.

The start column gives the *birth* date of the software project. In brackets we have set two projects where we could not clearly state its starting date. In the case of GNU Emacs, we can track the project until 1976, although it seems that Richard M. Stallman rewrote it from scratch in the mid 1980s. Mozilla, on the other hand, is the evolution of the Netscape Internet suite (which at the same time has a strong influence from Mosaic, one of the first web browsers). It is difficult then to know when the project really started, so the date in brackets gives the year of its first release under the terms of a libre software license.

The next column gives the date of version 1.0 for all those projects who have achieved that version number (which is valid for all of them but not for WINE). The third column gives the oldest line that we have found by means of our analysis. Dates in brackets can be found for those projects where the date of the oldest line is due to the initial check-in of the source code base into the versioning system. As the methodology is strongly tied to the versioning system we are not able to know what occurred before.

Project	Start	Vers. 1.0	Oldest line	SLOCs	SLOCCount	Percent.	Files	Authors
Emacs	(1976)	1985	May 85	974,407	991,552	98.3%	1,522	136
GCC	1985	1987	Sep 97	2,191,764	2,262,632	96.9%	22,349	218
Wine	1993	-	Oct 98	1,033,318	984,710	104.9%	2,201	2
GTK+	1994	Apr 98	(Dec 97)	387,413	389,723	99.4%	839	114
The GIMP	1994	Jun 98	(Dec 97)	548,410	552,473	99.3%	2,244	71
Apache 1.3	1995	Jun 98	Feb 96	82,909	85,758	96.7%	269	51
kdelibs	1997	Jul 98	May 97	605,528	613,742	98.6%	3,131	363
Evolution	1998	Dec 01	May 98	205,278	207,069	99.1%	816	79
Mozilla	(1998)	Jun 02	(Apr 98)	3,414,387	3,510,691	97.3%	19,604	567

Table 4.23: *Summary of the case studies. Columns contain the project name, the year the project started its development, the date of releasing version 1.0, the number of SLOCs according to our methodology, the number of SLOCs as given by another counting tool (SLOCCount) and the coincidence for both results. Finally authors that could be identified for the current version.*

The next three columns (SLOCs, SLOCCount and Percent.) give the number of lines of code that we have measured with our methodology (we filter the source code retrieved from the versioning system in order to obtain only source lines of code as described in our methodology), the number of source lines of code that SLOCCount gives for comparison and the percentage that this supposes. As we can see from the last of these three columns, the percentages are usually near but below 100% as our filter is not as powerful as the one used by SLOCCount and hence misses some lines. Only in the case of Wine we see a deviation which is meaningless. Finally, it should be noted that we have identified 374,523 lines that we have found in 2,667 files in Mozilla in Javascript that have been omitted from the table as SLOCCount does not count this language.

The last two columns in table 4.23 give the number of source code files and the number of authors (i.e. commiters) for these projects.

Definition of indexes

We will discuss in detail the software archaeology measures by means of graphical representations next, but first the definition of some indexes will provide numerical values that we can evaluate and compare. The indexes we propose are following:

- **Aging** (measured in SLOC-month). It is a direct measure of how much the software is aging and is given by the area under the curve of remaining lines which we will present next.

$$Aging = \sum_{n=1}^{N-1} lines_n \tag{4.3}$$

  where n is the month number, being n=1 the first month of the project and N the current one. Notice that the last month is not taken into account.

This measure is given in the sense of Parnas' well-known software aging concept [Parnas, 1994], although we only have in mind one of the factors stated by Parnas. If we would stick to Parnas' original definition of aging, then we should take into account changes performed on the system and not only that the software gets old as humans do. We lack of this information from the software archaeological point of view, but other methods could be implemented that extract the data required from a versioning system and set up a metric that fits the original definition more accurately.

In any case, our *aging* index may give an idea of how *old* a software system has become, but it does not provide much information about how much it has been maintained nor how easy it will be to maintain it in the future. On the other hand, the SLOC-month measures are difficult to be compared among projects as they depend on the size and the starting date of the software project. The latter problem can be easily solved by having a measure of *relative aging*.

- **Relative aging**. This measure makes it possible to compare the *aging* for several projects with independence of their size. It is measured in months and can be obtained mathematically from following equation:

$$RelativeAging = \frac{Aging}{lines_N} \tag{4.4}$$

  where N is again the last month considered.

The *relative aging* gives the amount of time necessary to have the same aging if the project had started with the current number of lines. It can be also seen as the amount of months needed to double the current *a*ge of the project if the system is not touched anymore.

- **Relative 5-year Aging**: relative size to itself as if the project had 5 years.

$$Rel5yA = \frac{Aging}{60 \cdot lines_N} \tag{4.5}$$

  where N is the last considered month.

Relative 5-year aging allows for easier comparison purposes, but also because it is a previous step for the *absolute 5-year aging* index which will be presented next. It should be noted that taking 5 or any other number of years is insignificant for the purposes of a relative index. In other words, this index is only meaningful when comparing two projects and for that situation the consideration of taking 5 years or any other number of years is thus not determinant.

- **Progeria**[7]. As *relative aging* gives the amount of time needed to double the *aging* value, we can compare it to the amount of time that it has taken to double the code base.

$$Progeria = \frac{RelativeAging}{50\%ofCurrentCode} \tag{4.6}$$

[7]Progeria is a genetic condition which causes physical changes that resemble greatly accelerated aging in sufferers. Source: WikiPedia

Values of progeria lower than 1 are indicative for software systems that are actively maintained and have not to fear the consequences of high values of *aging*, while values above 1 refer to projects where *aging* is growing more than software maintenance activity and therefore show a tendency to *dementia*.

Until now, we have only considered *aging* in order to find out how old the system is or how old it will get, but we have not found a value that tells us how maintainable the system is. The fact that for previous indexes we have taken relative size to the software size itself is confusing as systems that are by far larger in size may have values which make them appear younger than smaller systems. Of course, this cannot be taken as a proxy of the effort that a development team has to put when taking over the software maintenance process.

That is why we propose a new index that gives a value relative to a fixed-sized and fixed-time software system. This will enable comparison among projects.

- **Absolute 5-year aging**: gives the relative size as if the project had 100 KSLOC and had been started 5 years ago. Serves for comparison purposes among projects. The idea behind it is that the former *r*elative maintainability does not take the total size of the project into account, and thus may be misleading.

$$Abs5yA = \frac{Aging}{60 \cdot 100K} \tag{4.7}$$

  where N is the last considered Month

So far, we have considered only indexes related to source code, but software archaeology provides us as well with information related to the authors of the software. Indexes up to the moment considered that our team has not changed. This does not stick to reality, so we have to look at the evolution of the developers. In this sense, we will see in a later section a first approach towards the quantification of the half-life of the *core* team for some libre software projects (defined as the time required for a certain group of contributors to fall to half of its initial population).

One of the limitations of the following indexes is that the curves are not continuous which may suppose some undesired behaviors.

In the sense of the *aging* index we can infer from figure 4.28 a similar index for those lines for which their author has left the project.

- **Orphaning** (in lines-month): gives the amount of lines that prevail from developers that are not active anymore in the project multiplied by the number of months of inactivity by the developer.

$$Orphaning = \sum_{n=1}^{N-1} lines_n \tag{4.8}$$

  where n is the month number, being n=1 the first month of the project and N the current one.

As for the *aging* values, the *orphaning* values are difficult to compare among projects. We could calculate relative indexes as we have done for source code lines above, but as they depend on the same values, it is sufficient to have a *factor* that allows to do transformations.

- **Orphaning factor**: the number of lines-month given by the *o*rphaning index is by definition equal or smaller than the *a*ging of the system. It has to be seen as the fraction of old lines that belong to developers not participating anymore in the project.

$$OrphaningFactor = \frac{Orphaning}{Aging} \cdot 100 \tag{4.9}$$

  We multiply it by 100 in order to have it in percentages.

The orphaning factor may be seen as an index that gives how much of the existing code is supported by their original authors. Low values of the factor mean that the development/maintenance team has kept in time and is currently available in the project. This ensures continuity and is indicative for lower maintenance costs as if the software would be maintained by newcomers.

The first impulse would be to multiply this factor with the Abs5yA index, but this would give us a non-realistic vision, as this is as considering that the developers that compose our team do not have loss of memory.

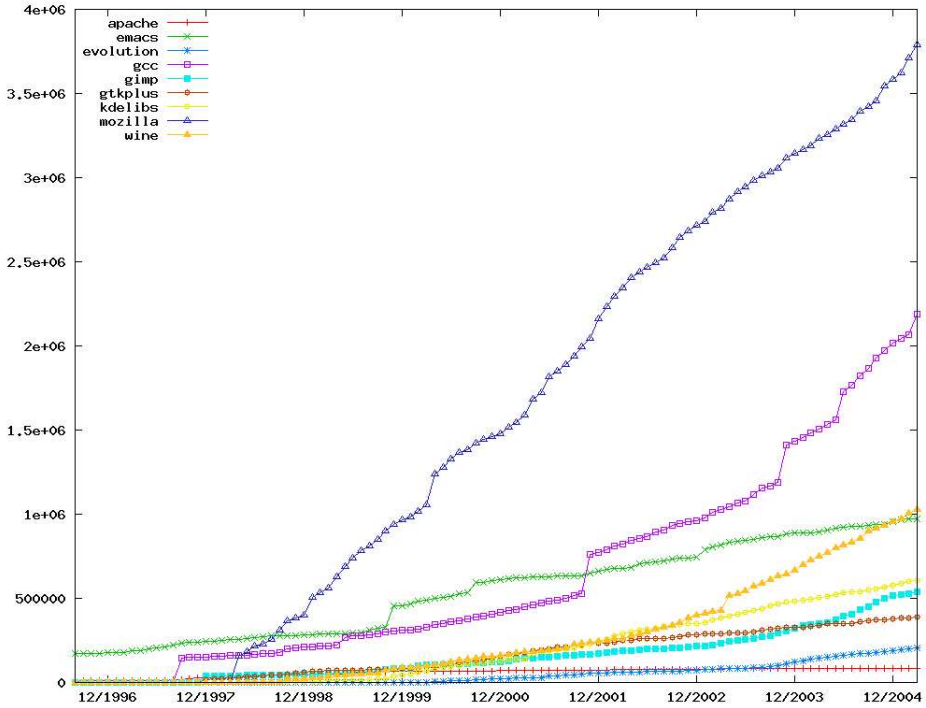


Figure 4.26: *Remaining lines (relative, aggregated values)*

4.4.3 Observations on the remaining lines

Figure 4.26 depicts the number of lines that remain from points in the past for all projects that have been considered in this thesis. In the horizontal axis we have time and in the vertical axis the amount of remaining lines for a given date is displayed. This is an aggregated figure in the sense that the figure gives the number of lines that are older than a given date. Unmaintained projects (i.e. projects with no changes) are characterized by a horizontal line. This is almost the case for Apache 1.3 (at the bottom of the figure), but not for the rest of the projects which show linear (as for instance Mozilla) or super-linear (for instance, Emacs) trends.

It should be noted that super-linearity in this figure does not mean that the project grows at that rate as it is usual in software evolution. The number of remaining lines is the sum of changed and added lines (archaeology), not only aggregated lines (evolution).

Figure 4.27 shows the same information as the one presented in the previous figure, but now the curves are relative to the size of the project. The horizontal axis is again time while the vertical axis is now measured in percentages. The current state of the project is given by 100% of the lines. We can read from this figure the amount of code that remains from the past relative to the current total size of the project. It may be seen as a proxy of the maintenance effort performed in recent times. Interestingly enough, projects lines are *younger* than we had expected. Besides Apache 1.3, all of them contain at least half of the code younger than 5 years, as it can be seen from Table 4.24. In that table, the points in time where projects achieve 30%, 50% and 80% of the current lines of code are given. Even the code base for Emacs, which we had selected as a presumably legacy system, has a major part (up to 70%) that is not older than 7 years.

The case of Apache is to some extent understandable as developers have focused on the new version of Apache (Apache 2.0). We expected that at least some corrective maintenance effort should take place, even if adaptative or perfective maintenance [Swanson, 1976] is not performed. But given these results, this seems to happen seldom.

4.4.4 Observations on the remaining contributions from authors

Figure 4.28 depicts the percentage of lines from remaining authors. Projects that do not lose their developers have more horizontal shapes. On the other hand, projects which have suffered high

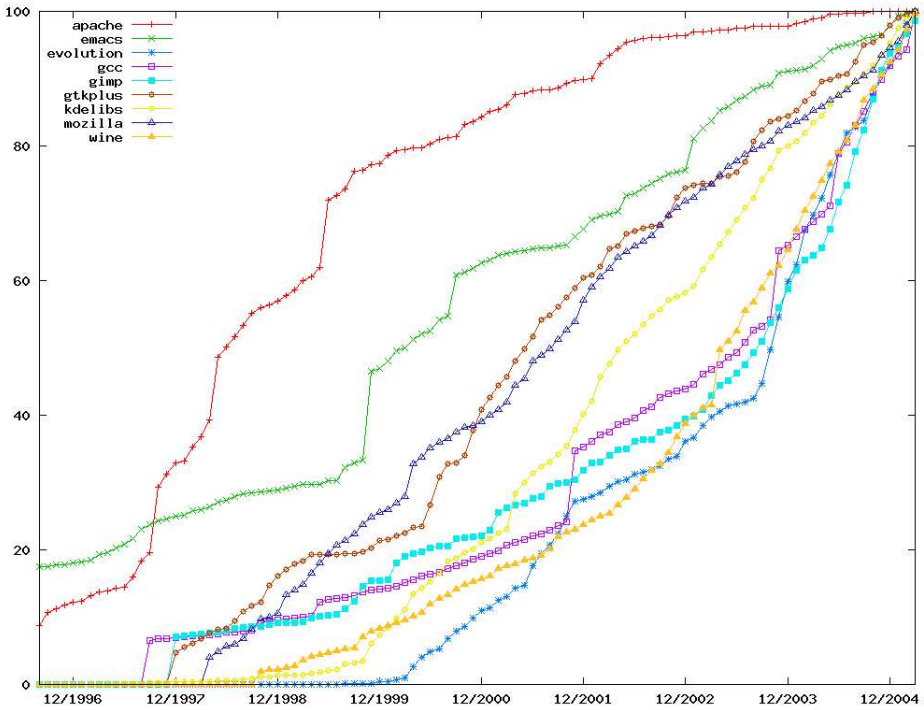


Figure 4.27: *Remaining lines (relative, aggregated values)*

Project	30%	50%	80%
Emacs	Jun 99 (69)	Apr 00 (59)	Jan 03 (26)
GCC	Nov 01 (40)	Jul 03 (19)	Jul 04 (8)
Wine	Jul 02 (32)	May 03 (21)	Jul 04 (8)
GTK+	Jul 00 (56)	Jun 01 (45)	Aug 03 (19)
The GIMP	Nov 01 (40)	Sep 03 (17)	Sep 04 (6)
Apache 1.3	Nov 97 (88)	Jun 98 (81)	Jun 00 (57)
kdelibs	May 01 (46)	May 02 (34)	Dec 03 (14)
Evolution	Apr 02 (35)	Nov 03 (15)	Jul 04 (8)
Mozilla	Apr 00 (59)	Sep 01 (42)	Sep 03 (17)

Table 4.24: *Most significant points in Figure 2 (100% is March 2005). This table gives the month for which a portion of code persists in the current version.*

abandonment have to fear a loss of system knowledge and this could have an effect in the maintenance effort (and cost) of the software in the future. The idea behind this is following: it is not the same to maintain and enhance a system having the original authors in the team than with newcomers which have to embrace a software comprehension process in order to become productive.

The curves in this figure are not continuous as the ones shown in previous figures (see figure 4.27) as they are related to developers and the whole amount of lines contributed by them. Hence, if a developer with a big share of the total contributions leaves the project we observe a vertical line. This is, for instance, the case for Emacs in November 2003 when Richard Stallman (the original author) appears to have contributed for the last time. This also means that values have to be below the ones in figure 4.27 as remaining lines for a developer have the date of the last contribution of the developer and not their own date (which has to be in any case before the last contribution).

Table 4.25 shows that the authorship indexes should have into account the amount of work performed. It is not the same to have committed 10 KSLOC years ago and a couple of lines last month than the other way round.

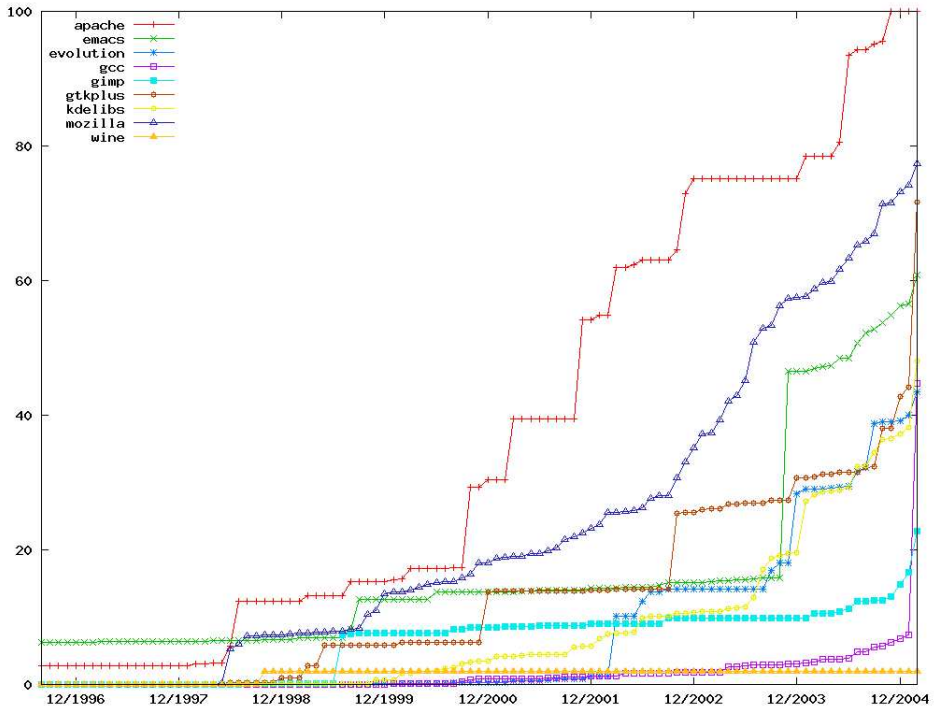


Figure 4.28: *Orphaned lines over time (relative, aggregated values)*

Project	30%	50%
Emacs	Nov 03 (16)	Jul 04 (8)
GCC	Feb 05 (1)	Mar 05 (0)
Wine	Mar 05 (0)	Mar 05 (0)
GTK+	Jan 04 (13)	Feb 05 (1)
The GIMP	Mar 05 (0)	Mar 05 (0)
Apache 1.3	Nov 97 (88)	Jun 98 (81)
kdelibs	Jul 04 (8)	Mar 05 (0)
Evolution	Jul 04 (8)	Mar 05 (0)
Mozilla	Oct 02 (29)	Jul 03 (20)

Table 4.25: *Most significant points in Figure 3 (100% is March 2005).*

4.4.5 Observations on the indexes

Once the indexes have been defined, we will apply them to our case examples (see table 4.26), and comment some particular cases.

From this table it can be seen how the aging index is not very useful for comparison purposes (although it gives a good idea of the absolute aging). However, relative aging allows for those comparisons. We can find a summary of the information in figure 4.27 in the corresponding column of table 4.26. Apache and Emacs are the systems with the highest relative aging. Evolution, Wine and The GIMP have values in the 20s, which mean that they are still actively maintained.

With respect to *progeria*, it can be said that it shows how Mozilla balances aging and evolution (this is the reason why it appears in Figure 4.26 as a linear function), while there are four projects which are becoming old systems: Apache and Emacs, but also GTK+ and kdelibs.

The absolute 5-year aging depends on the size, and has been presented as a proxy of maintainability. It yields that Apache, even having high progeria and aging is still more *friendly* to be maintained than the rest of systems (except for Evolution) because of its small size. Emacs and GCC, even having the latter two times the size of the former, have similar values, while GTK+ and GIMP also evidence this behavior.

While the orphaning index helps little in comparing projects, the orphaning factor provides very

Project	Size	Age	Aging	RelA	R5yA	Prog.	A5yA	Orph	OrFact
Emacs	974,043	239	62,419,261	64.1	1.07	0.93	10.40	18,585,711	29.78
GCC	2,188,033	91	65,558,122	30.0	0.50	0.65	10.93	3,427,811	5.23
Wine	1,028,820	78	26,926,319	26.2	0.44	0.80	4.49	1,521,212	5.65
GTK+	387,333	88	16,938,898	43.7	0.73	1.04	2.82	5,218,899	30.81
The GIMP	540,540	98	16,002,332	29.6	0.49	0.59	2.67	3,595,296	22.47
Apache 1.3	82,909	110	6,161,847	74.3	1.24	1.10	1.03	3,290,623	53.40
kdelibs	604,888	95	20,089,807	33.2	0.55	1.04	3.35	5,023,152	25.00
Evolution	204,951	99	4,796,800	23.4	0.39	0.66	0.79	1,665,455	34.72
Mozilla	3,786,735	84	161,394,929	42.6	0.71	1.00	26.90	90,902,668	56.32

Table 4.26: *Archaeology indexes for our case studies. Size is given in SLOC, Age in months, Aging and Orphaning in SLOC-month, Relative Aging in months, Progeria, R5yA (the relative 5-year aging) and A5yA (the absolute 5-year aging) are indexes and the Orphaning factor (OrFact) is given in percentage.*

useful information. It gives small values for GCC and Wine, very high values for Apache and Mozilla, and medium values (around 25% to 35%) for the rest of systems. This may give an idea of how much experience the current team has with the system.

4.5 File-type-based analysis

Software systems have evolved during the last decades from command-line programs to huge end-user applications full of graphics and multimedia elements. Besides, a piece of software has nowadays to be adapted to different cultural environments (language and notational conventions) if it aims to become mainstream. All this has caused that software development is an endeavor that is no longer carried out only by software engineers. In many cases it has become an activity that requires the coordinated work of several groups, with different backgrounds and that perform different tasks such as internationalization and localization (from now on `i18n`, short for *internationalization* and l10n, short for *localization*[8]), graphic design, user interface design, writing of technical and end-user documentation and creation of multimedia elements.

During the software construction process these diverse elements are handled together, conforming an integral body that has to be developed, managed and maintained. Despite this new environment, *classical* source code analysis is still focused on the output of the work performed by software developers: source code written in a programming language. The rest of the elements mentioned above are usually not considered, even though they are in many cases a fundamental and non-trivial part of the application.

In this section, our intention is to provide some insight into all those elements that conform a modern end-user software system. We propose a methodology for such a study, based on a software tool that implements it in an almost-automatic way. Our assumption is that many *traditional* concepts from software engineering may be generalized for other artifacts. For instance, in the same sense in which source code suffers from *software aging* [Parnas, 1994], all these artifacts have to be updated and handled conveniently to avoid similar effects. To accomplish this goal, we identify the several kinds of files found in a versioning system, mining in its historical information database for different patterns and behaviours. From the analysis of the files and their evolution we may not only infer the importance that a given software project allocates to the various activities, but also many other facts that may provide comprehension of the project and the whole development process. We will do this from several perspectives, focusing both on technical and human-related aspects.

4.5.1 Goals

The main goal of this section is to see how we can generalize *classical* concepts, like software evolution, to other elements in the sources different from source code. Therefore, we will first have a look at the relative importance (in number of files and measuring the activity) of the various file types that we have used in our classification. Then, we will see if the growth pattern (measured this time in number of files) is similar to the one found for source code files.

We will also be interested in finding if different behaviours arise for different file types. In this sense, we will look at the number of developers that *touch* a file for any given file type (we have called this analysis *territoriality analysis*) or have a look at the number of remaining lines for each file type (an *archaeology* analysis as we have already presented in section 4.4, but this time generalized to all file types and not only source code).

Besides technical issues, we will research social links. In detail, we are interested in knowing if commiters are specialized (i.e. devote their activity to a single file type) or not. As we have access to longitudinal data with our methodology, we will research the specialization of commiters in the first phases of the project and in the last ones and interpret what has changed if something has changed. We expect commiters to become more specialized while the project grows in software size and in number of contributors.

Finally, we will try to identify several *communities*, one for each file type. The members of the communities are the developers that have been more active in the given file type. Then, we will see how the members of different communities relate one to each other, finding developers that work on several activities. Especially interesting in this case is to see if developers and documenters work

[8]Internationalization is the process of designing applications so that they can be adapted to various languages and regions without engineering changes. Localization is the process of adapting software for a specific region or language by adding locale-specific components and translating text.

tightly together. On the other hand, we are also interested in the relationship between translators and documenters.

4.5.2 Methodology

The methodology described in this section is based on the analysis of the log entries from versioning repositories. We have automated the analysis with the CVSAnalY (see section 3.3 for further details).

For our purposes, we have classified, with the help of some heuristics, files into a set of file types, a method presented in subsection 3.2.2. Files that do not fall into any of the previous categories (generally those without extension or with an infrequent extension) have been labeled as *unknown*. Our experience with several large libre software projects with hundreds of thousands of files is that usually through manual inspection the amount of commits performed to unknown files can be minimized to values under 6% so that their effect is relatively small.

In addition, there is some file-specific information in the CVS logs that can be skimmed, such as whether a given file has been removed[9].

Once the CVS logs have been parsed and the file names are identified and sorted into the corresponding file type, the resulting data is fed into a database. Later this database is queried for patterns and behaviors for the selected file types, and for the commiters that have worked on them.

Since we are interested in the behavior of distinct file types in the repository, we intended to classify one to one atomic commits (ACs) with file types. This is, of course, possible if ACs contained only a single file type which has not to be the general case. This is the reason why we figured out the concept of predominant file type for a given AC. The idea is very simple: we attribute the AC to the file type that appears most frequently in it; in the case of having two or more file types that are equally represented, the assigned file type is selected randomly among them.

4.5.3 Case study: KDE

The K Desktop Environment (KDE) is a multi-million source lines of code (SLOC) libre software project, aimed to build a software graphical desktop environment for UNIX-like operating systems. Besides the window manager and desktop facilities, it offers as the KDE distribution an application development framework and a great number of applications that range from the KOffice office suite to the KDE-games game package. It is mainly written in C++, although some other programming languages are also used. The desktop and its applications are built by making use of their application development framework, the Qt toolkit. A large community has flourished around KDE and the number of commiters is over one thousand. The inner functioning, decision structure and organization of KDE is similar to the one described in [Germán, 2004b] (although that study was performed on GNOME, another libre software desktop environment similar in goals, size and technological complexity).

Table 4.27 gives a brief summary of KDE, its size in source lines of code, number of modules, files, commiters, commits, atomic commits, lines added and removed[10]. We have added the time of the first commit (when the repository was set up) and the one for the last commit considered in this study to show that our study considers eight years of development from the beginnings of KDE to its most current state.

We have applied our analysis to the whole KDE CVS repository, although we could have limited our research to a smaller granularity level such as CVS modules or others (such as applications or directories). In the KDE case, a CVS module may contain an application or a set of applications. For instance, there is a 750 KSLOC KOffice module that groups some office applications (word processor, spreadsheet and presentation program, among others).

[9] In CVS there is actually no file removal: files that are not required anymore are stored in the Attic and could be called back anytime in the future. But it can be tracked when one file is only in the Attic.

[10] Note that through this thesis we have considered SLOC as "a line that finishes in a mark of new line or a mark of end of file, and that contains at least a character that is not a blank space nor comment", in opposition to lines that consider not only comments and blank lines in source code, but also, for instance, text lines in documentation files

Software size	8,134 KSLOC
Number modules	90
Number files	442,445
Number commiters	1,163
Number commits	6,790,240
Number atomic commits	480,897
Lines added	324,925 K
Lines removed	297,294 K
First commit	1997-04-09
Last commit	2005-04-21
Number of days	2,934

Table 4.27: *General statistics for the KDE project.*

The reason for this can be inferred from figure 4.29 which presents the distribution of commits per module for the selected file types (both axes are in logarithmic scale), being development the sum of code, build and devel-doc file types. We can see from this figure that not all modules contain files from all the file types considered in this study (the point where the curves cross the horizontal axis is indicative —in logarithmic scale— for the number of modules that contain that specific file type). On the other hand, some modules contain a high amount of translation files, documentation files or images, a fact that can be read from the place where the curves cross the vertical axis. Further inspection of these modules has led to the conclusion that they do not contain a big share of other elements and that their purpose is project-wide. We have found that some modules serve for the project's own administrative tasks, and that there also exists a module that stores all translation (`i18n`) files. Hence, for a complete analysis of the file types of KDE the whole repository had to be considered.

Regarding the shape of the curves in figure 4.29, we had selected a log-log axis in order to identify the type of distribution. We expected to find power law distributions, common in other dynamic and social systems such as computer networks [Albert *et al.*, 2000], as they are among the scaling laws that describe a fractal growth behavior. However, the curves point out to follow Poisson distributions, which differ qualitatively from power laws. At the time of this thesis, we cannot indicate if such a shape is the consequence of coordinated management in the KDE project, and it would be interesting to investigate if software projects with no coordination at all result in power law distributions as we had assumed. Probably the study of other repositories, like the one of the GNOME project (which should yield KDE-like results, i.e. Poisson distributions) and of all the projects hosted at SourceForge.net (where there is no joint management effort, i.e. presumably resulting in power laws) would provide some insight into this issue.

Basic statistics

Entering into detail, table 4.28 sums up the main statistics for the KDE CVS repository: number of files, commits and predominant atomic commits for the distinct file types. These figures may provide an idea of the activity around any given file type that we are investigating.

A first impression offers some interesting information. KDE is clearly a software development project (code is the largest portion in the pies in figures 4.30 and 4.31). But the effort invested into development only reaches around 50% if we consider atomic commits as a measure of activity and around 25% if we take the number of files into account. The amount of translations is also a good indicator of the widespread adoption of the KDE project around the globe. Documentation and images are also heavily represented. The results also show how the number of multimedia files is minimal (in some sense denoting that KDE is not content-driven), while the user interface fraction (around 15%) is large enough to properly argument that it is actually a desktop-targeted environment. Finally, the share of atomic commits corresponding to the unknown file type lies under 3%, although they suppose around 6% of the total number of files.

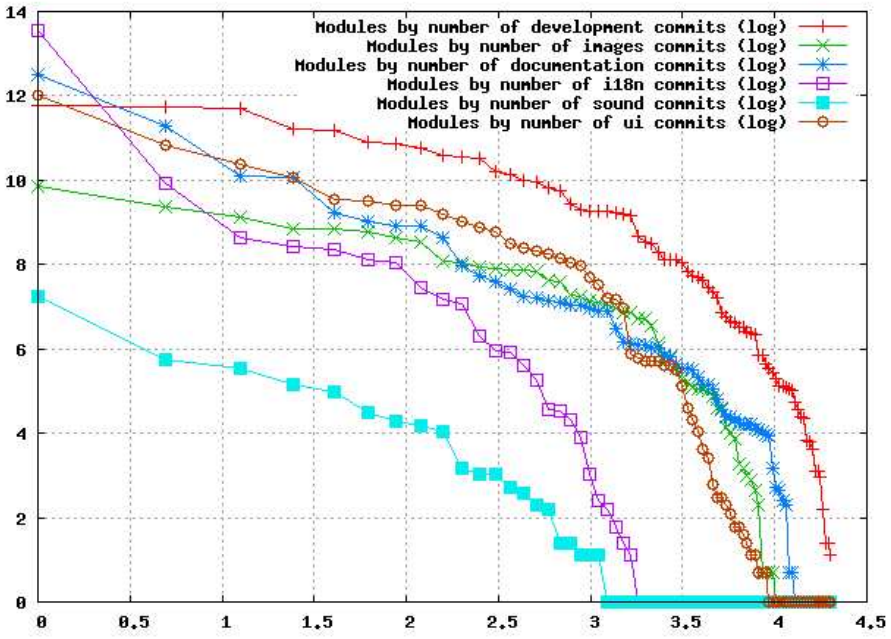


Figure 4.29: *Log-log representation of file types among KDE CVS modules. The vertical axis gives the number of commits, while the horizontal axis shows the number of modules. Modules have been sorted by number of commits, so those with higher number appear nearer to the origin.*

Another characteristic that we can infer from table 4.28 is that there exists some specialization among the commiters that devote their time to KDE (see column 'Commiters' and its share). This can be seen from the fact that none of the file types has been *touched* by all commiters. The most 'popular' file type is code with over 80% of the commiters ever having commited at least one code file, while multimedia files have been handled by just a few. The rest of the values, besides `i18n`, lie from more than 50% to some less than 80%. Commits to translation files can be observed only in one of every three commiters.

We have to point out that there exist severe differences if we consider the data from the point of view of single commits or do it by means of grouping them into atomic commits. This is especially the case for the `i18n` file type that drops down from an almost 60% share in number of commits to around 25% in the number of atomic commits. We will devote the next paragraphs to a detailed analysis of atomic commits where we will gain some insight into this circumstance.

File Type	Files	%	Commits	%	Pred.At.Com.	%	Commiters	%
All file types	442,445	100%	6,790,240	100%	480,897	100%	1,163	100%
Documentation	67,395	15.2%	546,487	8.0%	41,266	8.6%	692	59.5%
Images	88,901	20.1%	174,881	2.6%	8,807	1.8%	606	52.1%
`i18n`	83,415	18.9%	4,045,496	59.6%	123,566	25.7%	399	34.3%
`ui`	19,144	4.3%	546,443	8.0%	19,903	4.1%	754	64.8%
Multimedia	2,354	0.5%	4,703	0.1%	200	0.1%	88	7.6%
Code	107,855	24.4%	1,074,018	15.8%	231,785	48.2%	974	83.7%
Build	39,337	8.9%	203,298	3.0%	31,217	6.5%	898	77.2%
Devel-doc	8,443	1.9%	65,451	1.0%	12,242	2.5%	741	63.7%
Unknown	25,602	5.8%	129,463	1.9%	11,911	2.5%	744	64.0%

Table 4.28: *Basic statistics on the KDE repository by file type: number of files (and share), number of commits (and share), number of predominant atomic commits (and share) and number of commiters (and share).*

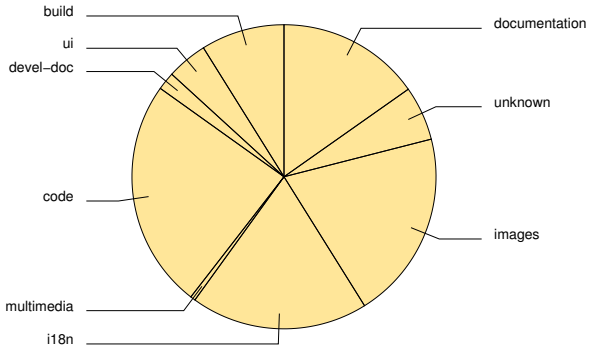


Figure 4.30: *Number of files by file type in KDE.*

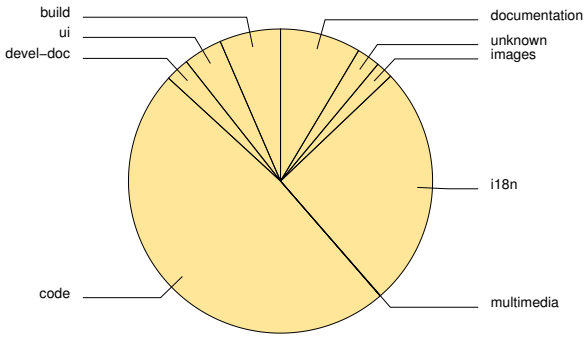


Figure 4.31: *Number of atomic commits by file type in KDE.*

Atomic commits

Atomic commits may provide an interesting point of view of the change patterns that occur in a repository. In this sense, we could look for coupling among source code files by identifying those files that are always committed together as done by Gall et al. [Gall *et al.*, 1997], denoting an inefficient system architecture if this behavior arises frequently. Similar concepts could be found for other file types or even among file types (for instance, if changes in the code are introduced simultaneously into the documentation). In the following paragraphs we want to further understand atomic commits looking for evidences that we may infer from them.

Table 4.29 gives the distribution of number of files that belong to atomic commits. In the case of KDE, most of the atomic commits contain a unique file. From previous research works, we know that this means that the use of a changelog file that is updated at the same time as changes are introduced is not common practice in KDE [Germán, 2004a].

Although being the most frequent, atomic commits with one unique file only group around 3% of all the commits done to the repository. We can see that there exist many atomic commits with many files, which seems surprising at first. For instance, atomic commits that affect more than 1,000 files correspond to only 0.2% of the total number of atomic commits, but almost to one out of every five commits. A more detailed analysis of the logs shows that this type of atomic commits belongs mainly to administrative and related tasks, such as checking in an initial version of the software, changing the year in the copyright notice, releasing software versions and moving, copying or removing a large number of files. We can conclude that the inclusion of atomic commits has as its first side effect that it filters out noise in our analysis, making the importance of the changes that have been introduced more conform with the actual change patterns of the project.

As pointed out in the subsection devoted to the description of the methodology (see 4.5.2), we have sorted atomic commits by means of identifying the most predominant file type found in them.

# files	Atomic commits	%	Commits	%
1	207,533	43.2%	207,533	3.1%
2	90,506	18.8%	181,012	2.7%
3	34,524	7.2%	103,572	1.5%
4	28,444	5.9%	113,776	1.7%
5	15,357	3.2%	76,785	1.0%
> 10	60,805	12.6%	5,777,911	85.1%
> 25	26,782	5.6%	5,224,383	76.9%
> 50	15,745	3.3%	4,839,605	71.3%
> 100	10,030	2.1%	4,432,171	65.3%
> 500	3,153	0.7%	2,937,576	43.3%
> 1,000	878	0.2%	1,294,392	19.0%

Table 4.29: *Distribution of the number of files per atomic commit.*

Predominance	Atomic commits	%	>5	(>5) %	>50	(>50) %
All	442,445	100.0	104,533	100.0	15,745	100.0
= 100%	405,096	84.2	64,493	61.7	9,911	62.9
< 90%	66,085	13.7	30,324	29.0	2,351	14.9
< 80%	51,627	10.7	19,030	18.2	1,611	10.2
< 60%	26,774	5.6	7,811	7.5	780	5.0
< 50%	5,305	1.1	2,877	2.8	340	2.2

Table 4.30: *Predominance of a file type in atomic commits. For all atomic commits, for atomic commits with more than 5 files and for atomic commits with more than 50 files. The data should be read as following: the first row gives information about all atomic commits, the second row provides the number of atomic commits where all of them (100%) are of the same type, the third row the number of atomic commits having less than 90% of the files of the same file type, and so on.*

Table 4.30 tries to give further insight into this process for our case study; especially for auditing if this classification can be treated as accurate enough. In this sense, we can see that almost 85% of the atomic commits are composed of file types of the same sort (see row =100%). Of course, this number comprises all the atomic commits with one file (which sum up to 43% of all the atomic commits).

In order to avoid the effect of having high percentages because of a reduced number of files in an atomic commit, we have considered in separate columns those atomic commits that group more than 5 files (which sum up to 104,533 atomic commits) and more than 50 files (of which we have 15,745 atomic commits). For these medium-sized and large atomic commits having all files of the same file type is uncommon, but the figures from table 4.30 give evidence that the attribution to a single file type has a high validity. So, only 10.7% of all atomic commits do not have a predominant file type that supposes at least 80% of its files (for atomic commits with more than 5 files the percentage is 18.5% while for atomic commits with more than 50 files it sinks to 10.2%).

In subsection 4.5.3, we have seen that some file types have a high number of commits, but a relative small number of atomic commits. Our first impression is hence that atomic commits for some file types are more prone to include a large number of files. Therefore we have computed the figures shown in table 4.31. We have considered only atomic commits with many files, discriminating them by their predominant file type. All those atomic commits that contain more than 50, 100 and 1,000 files have been selected. `i18n` is the most represented file type in all categories, a result that is consistent with previous findings. The results in the table show that atomic commits that group many files are very common for `ui`, multimedia and images, while the software development file types (code, build and devel-doc) have very low shares. Documentation lies in between of these two groups, being close to software development file types if we consider atomic commits with more than 50 files, but having the highest frequency of all file types for atomic commits that affect more than 1,000 files.

Code files grouped together in an atomic commit could be a good indication of (too much) common coupling, although in order to assure it we should look if changes to files coincide frequently [Gall *et al.*, 1997]. The numbers for documentation are not surprising as many documents are dispersed in

File type	>50 files	%	>100 files	%	>1,000 files	%
All	15,745	3.3%	10,030	2.0%	878	0.18%
Documentation	967	2.3%	578	1.4%	207	0.50%
Images	518	5.9%	235	2.7%	15	0.17%
`i18n`	9,476	7.7%	6,559	5.3%	593	0.48%
`ui`	2,152	10.8%	1,626	8.2%	22	0.11%
Multimedia	24	12.0%	12	6.0%	0	0.00%
Code	2,009	0.9%	739	0.3%	21	0.01%
Build	368	1.2%	166	0.5%	6	0.02%
Devel-doc	17	0.1%	7	0.1%	0	0.00%
Unknown	214	1.8%	108	0.9%	14	0.12%

Table 4.31: *Number of atomic commits for each file type that affect >50, >100 and >1,000 files. The share gives always the fraction of atomic commits that affect >50, >100 and >1,000 files related to the total number of atomic commits per file type.*

several XML files and are worked on by a single commiter at a time. So, a frequent situation is to commit all these documentation files once at a time together resulting in atomic commits with many files. In the case of `i18n` we have noticed that there exists a *gate-keeper* effect, as only a few translators have write access to the repository. Gate-keepers thus may wait until they have a considerable amount of (personal and third-party) contributions before introducing the changes into the repository. And finally for the case of user interface files, we have observed that `ui` has the highest frequency with 12% of all its atomic commits having more than 50 files. We think that this is because there is no central place where desktop-wide changes may introduced. So, changing some visual configurations affects many files, yielding a behavior similar to the one observed when source code has (too much) common coupling.

Interesting is also the high amount of large atomic commits in which build files are predominant as we would assume that these files are modified in conjunction with code. Having that many atomic commits that mainly involve build files is indicative for frequent changes in the build procedures. On the other hand, this means that there may be also a high common coupling among these files in the sense that if a makefile has to be modified due to the introduction of a new flag, many files have to be *touched* and committed. A higher level of abstraction for this task should be desirable, although this probably means migrating to a new build system or surpassing the limitations of current tools with a better design.

Evolution

Next, we will apply Lehman's software evolution methodology to the file types that we are investigating, especially in regard to the growth of the software systems [Lehman *et al.*, 1997]. As Lehman used source code files[11] as the basic unit to measure software evolution this is an easy task with our methodology. There have been some previous studies on libre software from the software evolution perspective, being especially important a work by Godfrey et al. that showed a super-linear growth for the Linux kernel [Godfrey & Tu, 2000]. Such a growth does not comply with the *laws* of software evolution and has been labeled by Lehman as an anomaly, inviting for further research on this topic [Lehman *et al.*, 2001]. Results obtained for this thesis from the study of 22 large libre software projects demonstrate that the most frequent growth trend is linearity, being Linux an exceptional case (see section 4.2). All in all, even with a linear behavior, the performance of such systems seem to be superior than the one inferred from (or predicted by) the *laws* of software evolution [Turski, 1996].

Our goal is to find how file types perform over time and if there are substantial differences in their behavior. Therefore, for every file type we plot the evolution of the number of files in the repository (total), the number of files *in the attic* (i.e. that have been removed) and the actual number of files that somebody obtains when retrieving the current version of the project (delivered). Of course, the

[11] Lehman uses the concept of "module" in his writings to refer to a source code file. We will stick in this section to source code file in order to avoid the confusion with CVS module.

delivered number of files, the only of the three that Lehman took into account, is the result of the *total* files minus the *removed* files.

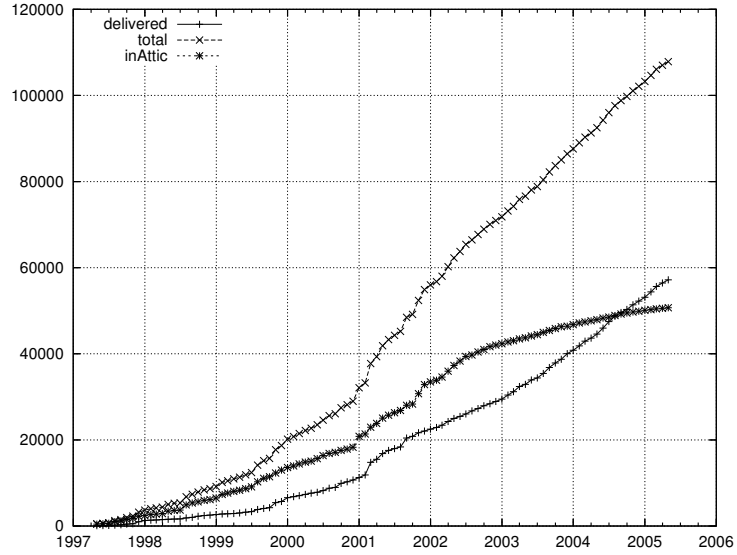


Figure 4.32: *Growth of the number of code files. The vertical axis gives the number of files, while the horizontal one gives the time.*

For the sake of brevity, we have included only two of the plots, one for the evolution of the number of code files (see figure 4.32) and another one for the evolution of `i18n` files (see figure 4.34) in a significant size and have summed up the rest in a smaller size in figure 4.5.3. Figure 4.32 shows a super-linear growth trend for code files in the early stages of the project for the three ways of counting modules that we have considered. From 2002 onwards, the number of total files shows a clear linear trend, but the number of files that have been removed clearly slows down yielding an accelerated growth pattern for the number of delivered files.

It should be noted that we are considering the whole project repository and that this means that we are really looking at an aggregate measure of many CVS modules (which may be at the same time the aggregation of several applications). In other words, the current behavior may be the effect of multiple independent development groups whose interactions are rare or at least limited. This was one of the arguments that Godfrey used to explain the super-linear growth of the Linux kernel: many parts could be developed and maintained independently [Godfrey & Tu, 2000]. This may be also the case for KDE, as the six KDE applications studied in section 4.2 had linear or near-to-linear behaviors[12].

Figure 4.5.3 depicts that all other file types follow similar patterns for delivered, total and removed files than the ones shown for code files. The results for multimedia files are the only ones that differ significantly from the rest as it can be seen from figure 4.34. For this file type, files are introduced and removed from the repository in a non-continuous way from time to time (with a rhythm that is similar to the main releases of KDE) and in large amounts, producing discontinuous curves. This may be explained in several ways. First, multimedia files (at least the ones we find in KDE, typically small sound effects) usually do not require the joint work of many people. We can imagine that there exists a high specialization for this type of tasks and will investigate this issue in subsection 4.5.3. And second, these files are mainly binary and CVS does not have a good support for those type of files; developers seem to avoid in these cases the use of CVS until it is finally necessary.

Territoriality

In a case study performed on source code files of a 200 KSLOC libre software application developed by a software company with the help of the *community*, a common pattern was observed and reported:

[12]The reader should note that the sum of linear behaviors may give super-linearity; this is because linearity is observed only once the project has started and not before. So, the aggregation of two linear projects results linear *iff* the projects have started at the same time.

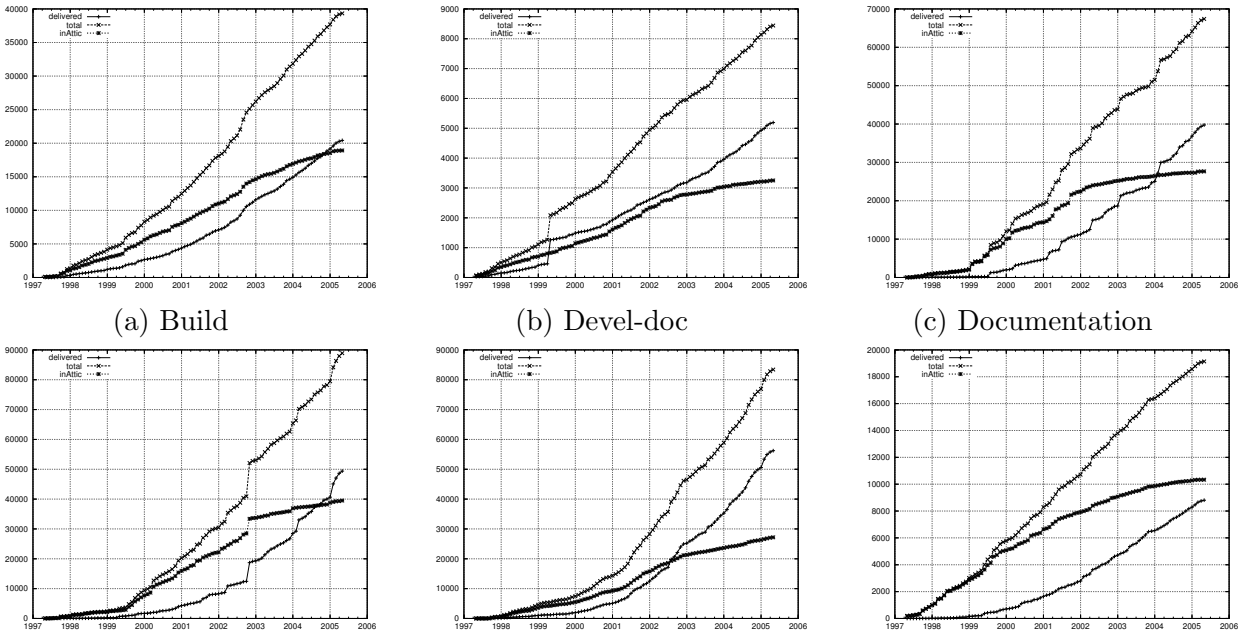


Figure 4.33: *Growth plots (delivered, total and removed files) for the specified file types. In all of them the vertical axis gives the number of files, while the horizontal one gives the time.*

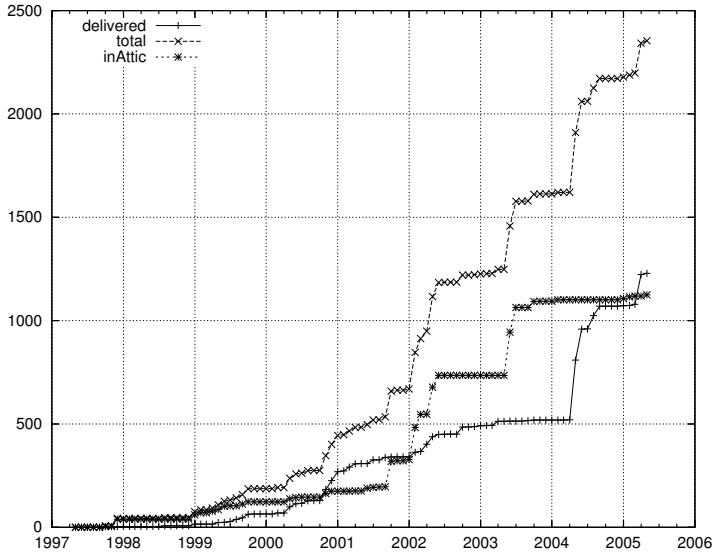


Figure 4.34: *Growth of the number of multimedia files. The vertical axis gives the number of files, while the horizontal one gives the time.*

that certain files were only *touched* by the same developer [Germán, 2004a]. This means, that even in the libre software field, where source code is publicly available and third parties are invited to join development, *code territoriality* may exist. Our intention is to check first, if this is also the case in larger environments such as the KDE project and second, if this is a general pattern for all file types.

Territoriality is of interest for maintenance purposes at least regarding source code (but probably for many other file types) for various reasons [Gîrba *et al.*, 2005]: first, it may allow to infer the organizational structure of the project, which may give some insight about the technical one [Conway, 1968]. Second, it allows to identify the original authors, supposed to have knowledge and experience on the underlying elements. And third, many commiters working on the same file could signify that peer review is done.

Hence, we have counted the number of commiters that have performed changes to a given file and have analyzed results statistically, discriminating by file type. The results are exposed in tabular form in table 4.32. Again, `i18n` seems to have a behavior clearly different from the rest. Almost all (if not all) translation files have been changed by at least two commiters. This is surprising as KDE usually does not provide write access to the CVS to all translators of each language, but only for a small set of them, and when previously studying the *atomicity* behaviour of commits we had found a gate-keeper effect. An inspection of the commits and the responsible commiters has given us the answer: some commiters (probably with the help of scripts) are responsible for a high amount of commits to `i18n` files, mainly for administrative reasons as copying and removing files. On the other hand, we can observe how the number of commiters in the last decile have very values compared to other text-based file types. This can be easily explained if we have in mind the diversity of languages into which KDE is translated; in this sense, we have many small teams that work independently on supporting a language. Finding a `i18n` file with many contributors will then be seldom the case.

The binary file types (mainly images and multimedia) present low number of authors, due in part to the limitations that CVS exhibits with this type of files. On the other hand, the need to change this type of files is usually lower than for the rest of text-based file types.

Regarding code, while the median value is two commiters, we have to wait until the forth decile to find files with two commiters. But interestingly enough, source code is the only file type that has a B80/T20 value[13] that is above 1 meaning that the weight of the two last deciles is not that significant as in all other cases (i.e. programming work is more equally distributed than it is for the other file types).

File type	Files	Mean	D1	D2	D3	D4	Median	D6	D7	D8	D9	D10	B80/T20
build	39337	2.39	1	1	1	2	2	2	2	3	4	86	0.81
code	107855	2.88	1	1	1	2	2	2	3	4	6	104	1.13
devel-doc	8443	1.9	1	1	1	1	1	2	2	2	3	52	0.81
documentation	67395	1.88	1	1	1	1	1	2	2	3	3	164	0.76
`i18n`	83415	3.37	2	2	2	3	3	3	4	5	6	22	0.57
images	88901	1.45	1	1	1	1	1	1	2	2	2	8	0.46
multimedia	2354	1.41	1	1	1	1	1	1	2	2	2	9	0.47
`ui`	19144	3.08	1	1	2	2	2	2	3	4	7	27	0.92
unknown	25602	1.7	1	1	1	1	1	2	2	2	3	278	0.64

Table 4.32: *Territoriality (in number of commiters) for files grouped by file type. The columns labeled as D* correspond to the deciles (the median is the 5th decile), while the B80/T20 column gives the results of dividing the sum of the bottom 80% with the top 20%.*

Documentation is a task that shows to have a very personal behavior, being the number of files that have been worked on by several commiters very low. As we have considered HTML pages as part of documentation, we expected these type of files to be changed frequently and by several persons. This may raise the question if enough effort is dedicated to this task in KDE. User interface files display a clearly different pattern with few files *touched* only by a single commiter (between 20% and

[13]The B80/T20 value gives a comparison of the contribution of the bottom 80% compared to the top 20%. If the B80/T20 value is 1, then we would have a situation where the bottom 80% contributes as much as the top 20%.

30%), but on the other side do not reach the high numbers that other text-based file types achieve in the last two deciles. In this sense, they stay in the same order of magnitude as `i18n`. This may be indicative for a small set of commiters having the required knowledge to change `ui` files. Some paragraphs below, we will try to find out if these commiters are mainly devoted to interface design tasks or if they are code developers that also modify the interface.

Finally, we can observe from the table that files that have been sorted as unknown do not have many commiters working on them. The results for these files shows that the mean value lies between the binary file types and the text-based.

File archeology

Software archeology may provide a useful metaphor of the tasks that a software developer has to face when performing maintenance on large software projects: the source code of a program at any point in time is the result of many different changes introduced in the past, usually by several people [Hunt & Thomas, 2002] (see also section 4.4). In this respect, the maintenance process can be thought in general to be more easy if affected files are not old (legacy) code. Our assumption is that files that are too 'old' (remain in the system unchanged from early stages in the development process) are less maintainable, since the developers that created them may not be part of the team anymore.

The archaeological point of view is conceptually very close to the idea of *code decay* as proposed by Eick et al. [Eick *et al.*, 2001] and tightly related to the software evolution results shown above. In the case of the super-linear growing Linux kernel, Godfrey stated that one of the causes for such growth could be the presence of many old files, especially drivers, that are not maintained but that are kept in the kernel as some users may still use them [Godfrey & Tu, 2000]. As we have seen, KDE also displays a super-linear growth trend for almost all of its file types, so an archaeological analysis may show if this is because of the presence of massive legacy code.

The methodology supporting archeology is very simple: we will look at the files that the current version has (so, we do not take into account those that have been removed) and find out when they were modified last (for more details, the reader may consult section 4.4).

Figure 4.35 is an aggregated plot of the files that remain unmodified since 2001; hence, the curves for all file types end April 2005 with their current number of files. We can observe from this figure that most file types show a similar trend having few files that were modified last in 2001 or earlier and an increasing number of files that have been created or modified in recent times.

Especially acute is the case for `i18n` files, where almost no files prior to mid-2004 are delivered currently. This means that translations files are updated continuously; the explanation for this behavior is that any new functionality in a graphical environment has a side effect on the menus of the applications. This requires to modify the files that contain the menu labels to be translated. Usually, labels are grouped at the application level in one file per language, so even one modification in a code file that changes a label implies many commits to `i18n` files. Of course, this provides another explanation of why the number of commits for `i18n` files is that large.

Figure 4.36 gives the same information than the one depicted in the previous figure, but this time the vertical axis is relative to the total number of files for each file type. Besides the already mentioned findings for the `i18n` files, we can see a curious effect for the devel-doc files, for which almost 20% still persist in the version that was introduced in the repository in the year 1999. We have inspected what provokes this behaviour and have found that it is due to the introduction April 30 that year of 853 .lsm files. These files are used as entries for the Linux Software Map, a project with the objective of listing all software available for Linux.

On the other hand, it should be noted that more than half of the files (of all file types) have been modified in the last two years. This may give an idea of the vivacity of the project as well as a good indicator of the maintainability in the next future as many of the commiters that performed the last modifications are probably part of the current team.

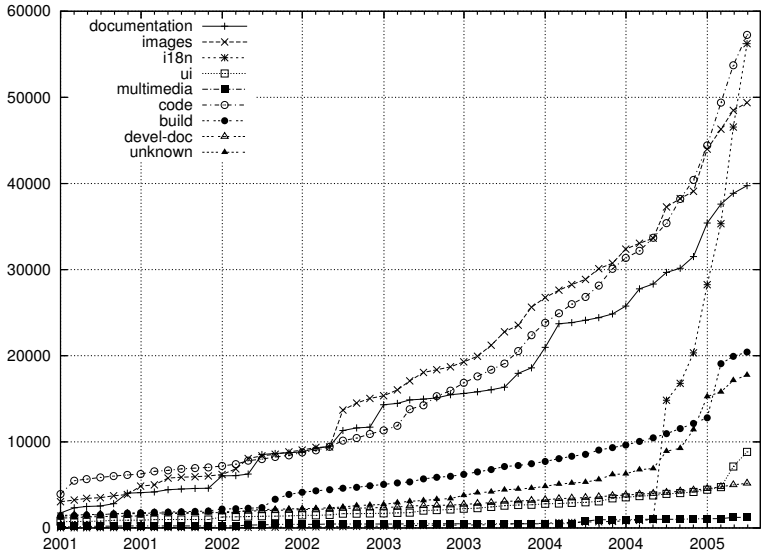


Figure 4.35: *File archeology for all file types. Absolute values in the vertical axis measured in number of files. The horizontal axis is given by time starting January 2001.*

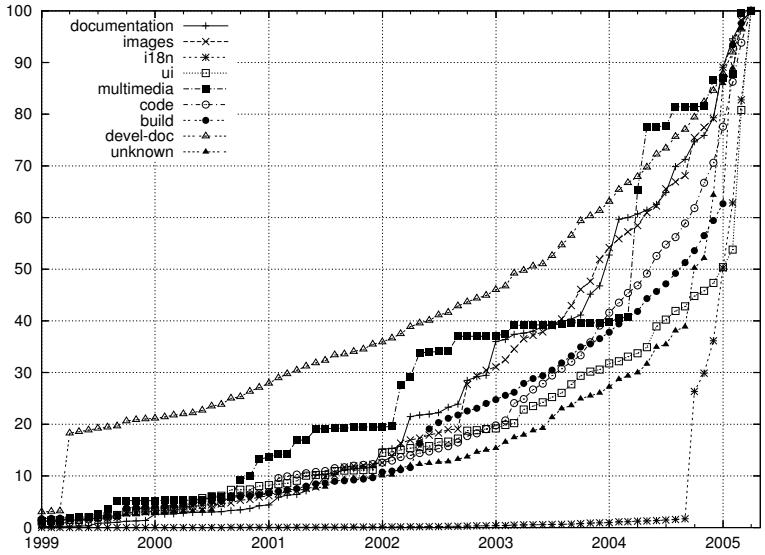


Figure 4.36: *File archeology for all file types. Relative values in the vertical axis measured in percentage (100% gives the files delivered currently). The horizontal axis is given by time starting January 1999.*

Commiter specialization

In libre software projects write access to the versioning system is usually not granted to everybody. This privilege is typically limited to contributors who reach a certain degree of commitment with the project and its goals. The relationship of the human resources with the file types and with each other deserves a study, especially in the libre software environment where developers and the tasks they devote their time to are self-selected and in general self-organized.

Heat maps may provide a visual idea of the specialization of commiters, showing at the same time the relationship that exists between file types. Therefore, we have selected those commiters that have worked on a given file type and compute the number of atomic commits they have performed on it. Then, for the same set of commiters we look at their contribution on other file types (again in number of atomic commits). The fraction of the latter divided by the former will give us the relationship. This idea can be summarized in following equation, where i gives the row, j the column index and $atomic\ commits_{j|i}$ means atomic commits done to j by active commiters for i:

$$Relationship(i,j) = \frac{\sum atomic\ commits_{j|i}}{\sum atomic\ commits_j} \tag{4.10}$$

If this fraction is close to 1 it will correspond in the heat map with a hot zone, represented as a dark color, while values close to zero will correspond to cold zones, represented by clear colors. In order to make the reading of the heat map easier, white has been reserved as background color and appears in the diagonal (that really should be black, since all its values are 1). Applying this idea on our case study will make the methodology clearer. In any case, it should be noted that the relationship shown in the heat maps has not to be symmetric, as in general $Relationship(i,j) \neq Relationship(j,i)$.

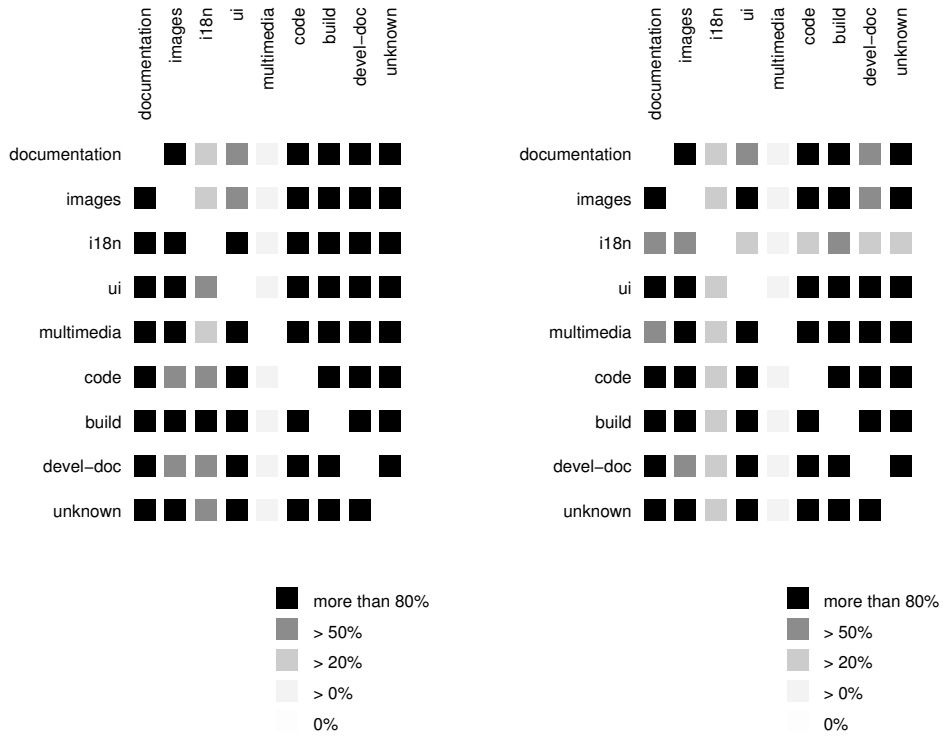
Since historical data for the project is available, we may also study the evolution of specialization. In this regard we have taken ten equally large time slots from the first commit to its present state, and have produced a heat map for each time slot. In the case of KDE, the first commit was done in April 1997, while the last one considered in this study dates from April 2005. Hence, the interval corresponding to each time slot lasts for about 293 days (almost 10 months) of activity.

Figure 4.37(a) corresponds to the time period from April 97 to February 1998, while figure 4.37(b) corresponds to the last time period (from June 2004 to April 2005). Examining figure 4.37(a) may illustrate how a heat maps work. The first row shows the relationship of documentation with all other file types considered in this study. As noted before, the intersection of documentation with itself has been left white. The second column shows the fraction of all atomic commits on documentation that corresponds to commiters who have contributed both to documentation and images. This is a very hot zone (with values that lie between 0.8 and 1). The next column provides information about the number of atomic commits to documentation by those committing files to documentation and `i18n`. The relationship here is mild, as it lies between 20% and 50%. The contribution of commiters to documentation that have also worked on the user interface (forth column of the first row) lies between 50% and 80% and on multimedia (fifth column) is very cold being between 0% and 20%.

In general terms, the most significant result from the analysis of the first time slot is the light color for the multimedia column. This means that we have found a specialized group of commiters who was responsible for the multimedia elements in KDE for the investigated time slot. We could point this out as an exception as the common pattern is to have dark (hot) zones, i.e. high degrees of relationship.

However, figure 4.37(b) shows that no such uniformity exists in the last time slot; lighter zones have become more frequent. In addition to multimedia files, we see that a new specialized group has emerged: translators (see `i18n` column). Also interesting is the fact that documenters do not have a big overlap with user interface and development documentation files, hinting to a trend towards becoming a more identifiable group in the next future.

In any case, we can infer without doubt from inspecting both heat maps that the KDE project has undergone a specialization process that has occurred in parallel to its growth in the last years. At the beginnings, contributors were active within all file types (with the exception of multimedia), while as the project has evolved and has become larger in software size and in number of developers, we observe a clear tendency to find contributors that have a specific task in the project. It should be interesting



(a) 1st time slot. (b) 10th (last) time slot.

Figure 4.37: *File type relationship heat maps for the first and tenth (last) time slots.*

to see how much this has been the effect of the investment that some software companies have made in KDE, especially in the sense of hiring already active developers or adding external man-power to the project.

The previous heat maps raise the question if different commiter communities coexist in the project, with members of each community targeting a specific task and thus working predominantly on a given file type. We have therefore come up with an idea in which we use scatter plots that allow us to see on a two-to-two basis how commiters behave for two selected file types. The distribution of commiters in the XY space will give an idea of the specialization of commiters as well as the possible relationships that may exist between file types.

In the scatter plots any point corresponds to a commiter. The shape of the point is given by the file type in which a commiter is being more active. Shape assignation follows the rules that are summarized in table 4.33. In order to make the data offered by the scatter plots easier to work with we have taken the natural logarithm of the commits done by commiters. On one axis of the scatter plots contributions to a given file type are shown, while on the other axis contributions to some other file type are displayed. Commiters with commits on both file types will appear somewhere in the first quadrant, while those who only have commits on any of the two file types will be on the respective axis.

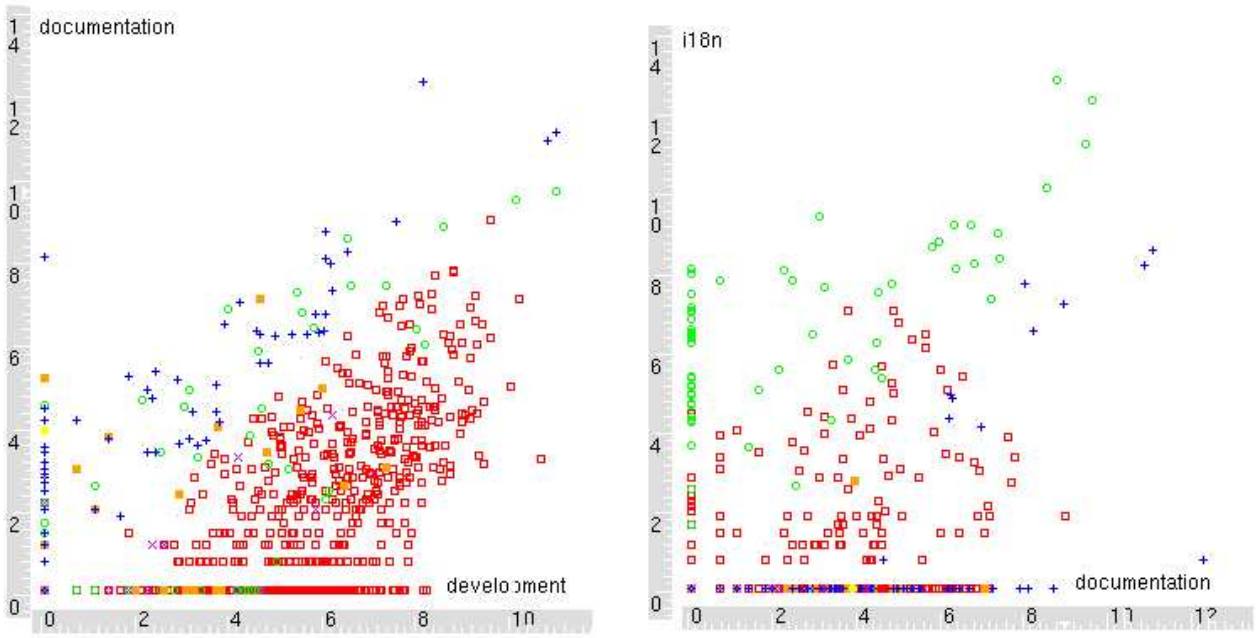
This means that only active commiters will be shown (those who have at least one commit in any of the two file types under consideration). This confronts us with the problem of commiters who have not done commits to one of the file types but with a notable contribution to the other. In order to have them included, we have considered commiters that have 20 or more commits[14] in one file type to have at least one commit to the other (if no commit had been done this was added automatically). This should not be a dramatic distortion of the data and should give us valuable information.

The first scatter plot that we will examine is shown in figure 4.38(a), and presents development (which groups the code, build and devel-doc file types) contributions in the X axis versus

[14] ln(20) is almost 3.

File Type	Shape
Development	open rectangle
Documentation	plus (+)
`i18n`	open circle
Images	filled rectangle
Multimedia	filled circle
`ui`	X
Unknown	point

Table 4.33: *Shapes used in the scatter plots that allow to identify the file type for which commiters have been more active (in number of atomic commits). Development groups code, build and devel-doc.*



(a) Documentation vs development (b) i18n vs documentation

Figure 4.38: `Community` *scatter plots. Number of commits of developers on the given file types. Axes in both figures are in logarithmic scale.*

documentation in the Y axis. There are several interesting facts that can be learned from this figure. First, that the development *population* (open rectangles) is by far larger than any other one. Second, the location of commiters whose primary task is none of the two depicted is also interesting: translators (open circles) are generally grouped with documenters (pluses), while those who work on the user interface (X) appear in the open rectangle development dust of developers. Third, we can find among the most contributing commiters ($\log(\text{commits}) > 10$, i.e. more than 20,000 commits) to the development file type nine persons, but only five of them have development as their first activity. Two more are mainly translators (open circles) and the other two are primarily documenters.

Figure 4.38(b) puts the documentation commits in the X axis and sets in the Y axis the ones related to `i18n`. It can be seen once more how there are several patterns followed by points of the same shape. It is interesting that the Y axis contains only open circles, which means that many commiters only perform translation-related tasks. From this fact again we can infer some degree of specialization in a project: almost half of the translators in the KDE project do not perform any documentation activity at all. Another interesting finding is that commiters whose priority are code commits appear usually mixed with documenters but not with translators, meaning that the affinity of these two groups may be bigger than with the people who do `i18n`.

To check if there was some correlation among the behavior of the different commiters, and whether there was some natural grouping of them, we trained a 15x17 self-organizing map (SOM) [Kohonen,

1990; 1995] on the test data, using standard training rules.

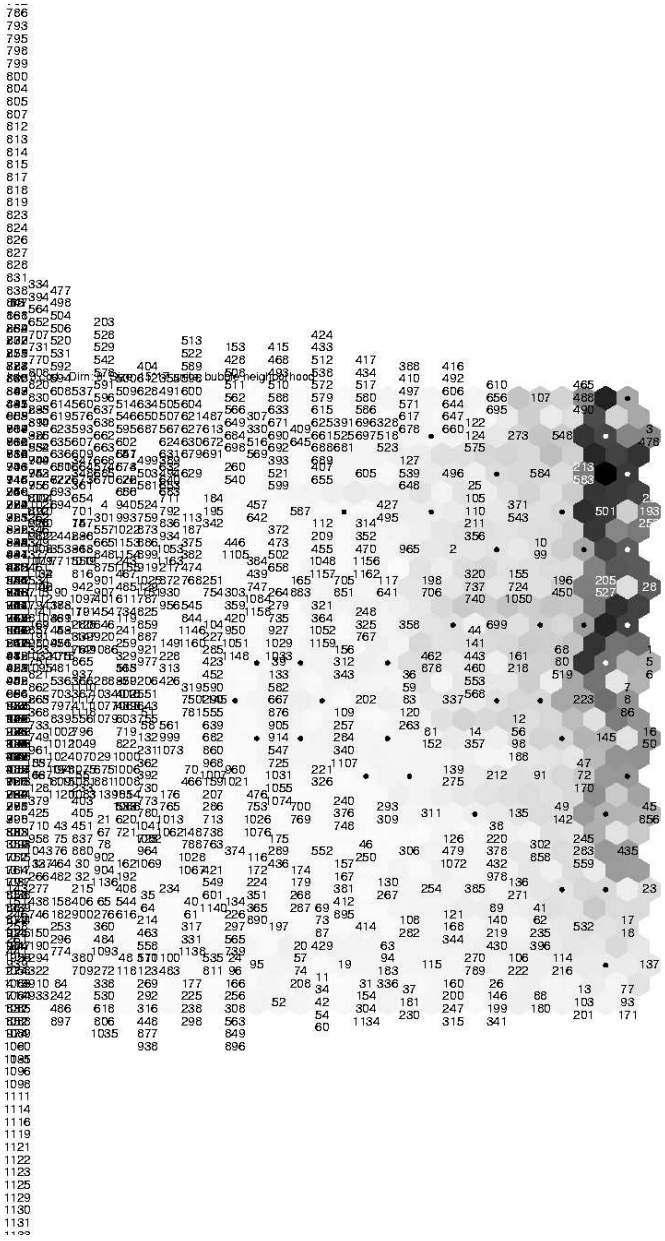


Figure 4.39: *UMatrix analysis on the trained self-organizing map.*

Figure 4.39 shows the result of running an UMatrix analysis [Ultsch & Siemon, 1990] on the trained SOM with developers given by their id number. This analysis highlights differences between the set of vectors that constitutes the SOM and makes clusters stand out. In this case, *clusters* are dark dunes surrounded by lighter ones, such as the one that appear in the right-hand side border of the map. There are four such zones, plus an undifferentiated set of vectors in the rest of the map; these four clusters are groups of developers that, somehow, stand out from the rest, probably due to their number of contributions.

If we want to inspect which contributions make these developers stand out, a planes analysis has to be run on the map. These nine maps in figure 4.40, that correspond to the same map as the one shown before, represent the value of the 9 components (corresponding to each of the file types considered in this study) of each vector; a light color corresponds to a high value of that component, and a dark color means a low value for that component). This demonstrates that the cluster at the top right corner is characterized by a high number of contributions to documentation, i18n and ui, and also a medium value of build requests. An explanation for this is that at least for this group which has a

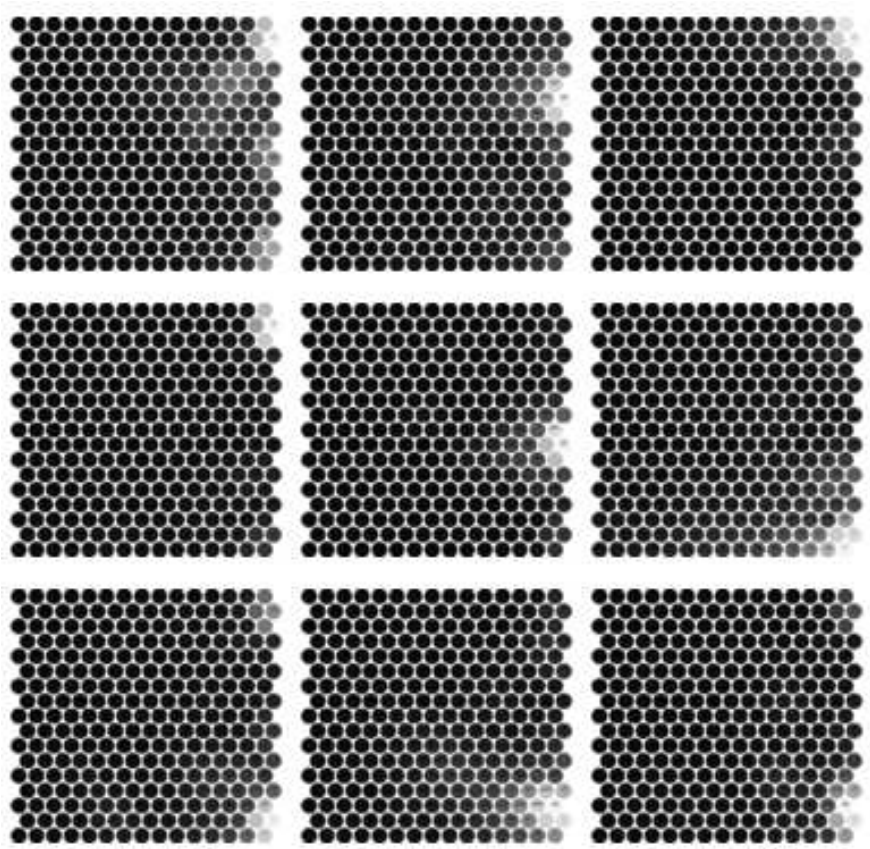


Figure 4.40: *Component analysis for all components. From left to right and from top to bottom: documentation, images, i18n, ui, multimedia, code, build, devel-doc and unknown.*

high number of i18n atomic commits, a high level of documentation and ui ACs, and a medium value of build ACs is given; this is not in contradiction with the rest of the studies, since these cluster only includes a few members with a high number of commits.

From the top right corner, the next cluster in the right-hand side is characterized by a high number of contributions to images, and the next cluster by many contributions to multimedia. Finally, the bottom right corner is formed by contributors mainly to code, build, devel-doc and unknown file types.

4.6 Social network analysis

Software projects are usually the collective work of many developers. In most cases, and especially in the case of large projects, those developers are formally organized in a well defined (usually hierarchical) structure, with clear guidelines about how to interact with each other, and the procedures and channels to use. Each team of developers is assigned to certain modules of the system, and only in rare cases they work outside their *territory*. However, this is usually not the case in libre software projects, where only loose (if any) formal structures are recognized. On the contrary, libre software developers usually have access to any part of the software, and even in the case of large projects they can move more or less freely from one module to other, with only some restrictions imposed by the common uses in the project and the rules on which developers themselves have agreed. A large mount of interaction structures arise, evolve and disappear without the intervention of a central control, yielding complex networks.

Among complex networks, social networks appear as a method for analyzing the structure and interactions of people and groups of people within extensive organizations [Newman, 2001a; 2001b; Guimera *et al.*, 2002; López & Sanjuan, 2002; López *et al.*, 2002]. To understand the structure of these networks, we are interested in determining how the different nodes interact and form groups that, in turn, interact with each other giving rise to higher order groups. The set of groups obtained, as well as their relationships, is which we call the community structure of the network.

4.6.1 Goals

The objective of this section is to apply classical social network concepts to the data extracted from libre software projects. This could be a first step towards gaining knowledge on the social structure of the project, which could be indicative of the technical one [Conway, 1968].

On the other hand, as libre software projects usually have a loose management style and are based heavily on third-party contributions, social network analyses may provide us with some insight into the underlying processes that are responsible for the development. An introductory study on the first versions of the Linux kernel (see subsection 4.6.3) will make this more evident in a visual manner.

But more complex networks, with a higher number of contributors and software artifacts, do not allow to extract information visually. We will therefore calculate a set of classic social network analysis indices (presented in subsection 2.4.2) that will provide us valuable information of the software projects selected as case studies in this thesis: KDE, GNOME and Apache. In addition, we have applied some structure-finding algorithms to the social links obtained from these projects. As we have longitudinal data, we will apply this algorithm to a project in several points in time and will try to infer some conclusions about its organizational structure and how (and if possible why) it has changed during the studied time interval.

Finally, we will test the networks of these projects to find out if a *small-world* behaviour is present, a characteristic that appears in networks with efficient information flows. We will attend also to other characteristics that can be of interest as redundancy or weights. Redundancy informs if the network is able to assume the loss of an edge (i.e. a developer) without suffering a great loss of efficiency. Weights on the other hand allow to discriminate taking into account the importance of the relationships; so, a strong relationship will have a higher weight than a loose one, probably yielding in different results (and interpretations).

As a summary, this section contains the application of traditional techniques from other fields of knowledge (the social sciences) to data obtained from libre software projects. This has been a fruitful research area, as already described in subsection 2.4.2 in the chapter devoted to related research. Our contribution is to show that the availability of data extracted from data sources and by means of methodologies described in this thesis allows to have a wider range of possibilities.

4.6.2 Methodology

Our methodology assumes that we can define two different kind of networks that characterize the organization of a software project:

- Developer network. Each vertex represents a particular developer. Two developers are linked by an edge when they have contributed to, at least, one common software artifact. Edges may be weighted by a *degree of relationship* defined for instance as the total number of commits performed by both developers on artifacts to which both have contributed.

- Artifact network. In this case, each vertex represents a particular software artifact (a software project, a CVS module, a directory, a file, etc.) of the project. Two artifacts are linked together by an edge when there is at least one developer who has contributed to both. Those edges are weighted using a *degree of relationship* between the two artifacts, defined for instance as the total number of commits performed out by common developers.

Relationships between software artifacts are considered when there are common contributors, assuming that their contributions require a high level of coordination. In other words, when two developers work on the same artifact, their actions have to be coordinated, and this coordination is considered a relationship. The corresponding network of developers is also studied; relationships are identified when developers have worked on the same module.

Several ways may be followed to obtain the relationships between developers and artifacts depending on how we define a relationship. In this section we will propose two approaches: a first one that will serve as an introductory part to social network analysis using early versions of Linux kernel as case study and a second one that studies large projects as Apache, KDE and GNOME.

In the case of the Linux kernel, we have scanned the Linux 1.0 version sources for authorship information with the help of the CODD tool (described in subsection 3.2.5). As a consequence of this process, we obtain a list of files and the names of the developers that have worked on these files. We have grouped files in directories (i.e. all files that belong to the same directory have been grouped) as usually the copyright information contains only one author per file even if more authors have *touched* that file. Considering all files in a directory compensates this. Based on these data we will build a developer network. Hence, developers (vertices) that have authorship attribution in files in the same directory (edges) are connected. The resulting graph will give us interesting information about the organizational structure of the project.

The other analysis considers not only developer networks, but also artifact networks. We have used as case studies large libre software projects that host a CVS versioning repository, and the software artifacts we have chosen are CVS modules. CVS modules are the highest-level directories in the versioning repository and usually correspond to software applications, libraries or software suites of related applications.

The process begins by downloading the relevant information from the CVS repository with the help of the CVSAnalY tool presented in section 3.3. This information includes, for each commit (modification in a file in the repository): the date, the identifier of the developer (commiter), and the number of lines involved. Using all those records, we define the developer and artifacts networks.

For the developer network, we consider commiters as vertices; a connection (edge) between two developers (vertices) exist if the developers have commits in a module in common. So, if two developers have worked in the `gimp` module of the GNOME CVS repository, they should be linked. The connections may be weighted, using as weight the the number of commits to the modules in common.

In the case of the modules network, modules are the vertices; connections (edges) between two modules (vertices) will appear if there are developers in common, i.e. at least one developer has performed commits to both modules. Again, we can define a weight for the connections. We have used as weight the amount of commits to the modules by developers that have collaborated in both of them.

Finally, we will analyze the developer and modules networks using traditional social network analysis concepts introduced in subsection 2.4.2: degree, clustering coefficient, distance centrality, betweenness and *small-world* behaviour. In addition, we will introduce the Girvan-Newman algorithm to study the structure of the project and comment the results.

4.6.3 Observations on the Linux 1.0 version

This subsection serves as an introduction to the possibilities that social network analysis offers for the study and comprehension of software projects. We have therefore taken the first mature version of the Linux kernel, released in 1994, as an example. We already know from the software evolution analysis of the Linux kernel (section 4.2) that Linux has become an immense system. But its 1.0 version is of smaller size, ranging around 100 KSLOC. This will allow us to perform our social network analysis visually; for larger programs the introduction of indices and other techniques will have to be used.

Figure 4.41 gives the relationship map among developers for Linux 1.0. Developers are displayed as vertices (given as red points). They are connected to other developers if they have authorship information in common source code files in the same directories. So, two developers who have contributed to the *arch* directory would be connected in this graph. It should be noted (as it has been shown in subsection 4.2.3) that at the time of Linux 1.0 the most important subsystems already existed, although of course not with that many subsubsystems created later.

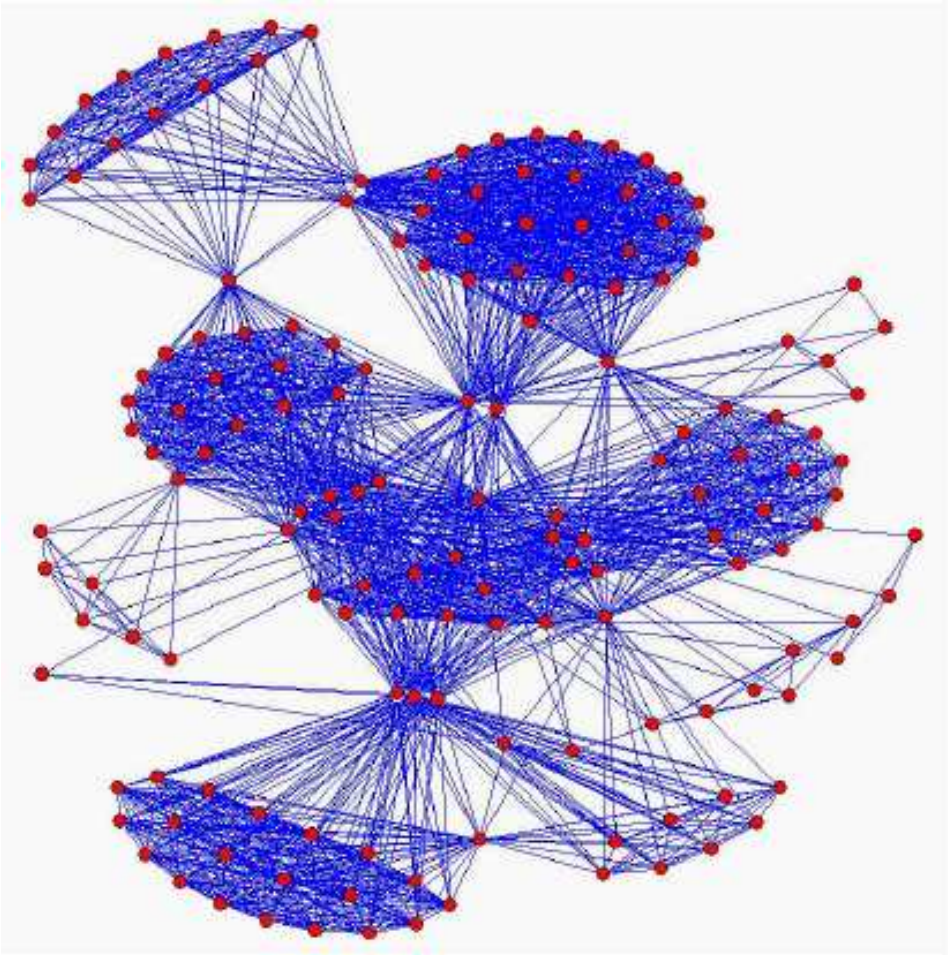


Figure 4.41: *Social network analysis (developer network) on Linux (version 1.0).*

The low number of contributors and the organizational structure that can be found in the Linux kernel for this version gives a clear visual graph. This graph is suitable to describe some of the main concepts of social network analysis. The first fact that we can observe is that we can clearly identify the existence of groups (clusters) of developers who work on the different parts of the Linux kernel. Clusters are formed by developers who are connected to the rest of the developers of that cluster. In social network analyses jargon they are also known as *cliques* and are indicative for communities with strong relationships and with high information flows.

The connection between two clusters is done via developers who have participated in both of them. This type of developers have a *linchpin* role, which a strategic position regarding among others

information control and dissemination. So the cluster in the upper left corner of figure 4.41 is connected by a *linchpin* developer to another cluster below. This developer servers as a bridge between these two groups formed by around ten developers each. His position is very important for the information exchange. In the case of the first cluster, *linchpin* positions are even more important as the cluster is connected to the rest of the project only by three developers. In their absence, this directory would have no connection to the rest of the project, at least for the definition of relationship that we have considered for this analysis, and be isolated.

Another characteristic of vertices in a social network analysis is their centrality. Several definitions of centrality exist. The degree is one of the most frequent, as we have seen in chapter 2.4.2. In our figure, we see that the vertices in the center are those who have a lot of connections to other nodes. One of them, the most centered and connected one, is Linus Torvalds. Torvalds is connected to many developers that are part of different clusters as we would expect from his role as project leader. But interestingly enough there are many clusters that are not directly connected to him, being this a consequence of modularization, i.e. work performed in parallel by distinct groups.

As we can see for this example, we can extract interesting information from the visualization of the network. But the problem of visual methods is that they are not scalable. We have taken as case study Linux in its 1.0 version because it is small in number of developers and of connections. But this is not the case for newer Linux versions and, in general, for the large projects that we will study in the next pages. For such large type of projects, the information that can be obtained with visual methods is very limited if not impossible at all. So, adopting the same methodology for newer version of Linux will not result in such a clear graph as figure 4.41, but in a chaotic mess of vertices and connections. If this is the case, other techniques are worth to be applied.

4.6.4 Observations on large libre software projects: Apache, KDE and GNOME

Apache, KDE, and GNOME are all well known libre software projects, large in size (each one well above the million lines of code), in which several subprojects (modules) can be identified. In the next pages, we are going to use these projects to show some of the features of our proposed methodology for applying social network analysis to software projects.

As already described, the first step for our analysis is to download information from the corresponding CVS versioning repositories, and to use this information to build the modules and commiters networks. Tables 4.34 and 4.35 summarize the main parameters of both type of networks. In the case of commiters networks the GNOME case has been omitted. In contrast to the Linux 1.0 version (see subsection 4.6.3), we are now managing a larger amount of vertices and edges for both developer and modules networks that hinder a visual approximation to the analysis of these networks.

Project name	Modules (Vertices)	Edges	Average	% of edges (avg)
Apache	175	2491	14.23	8.13
KDE	73	1560	21.37	29.27
GNOME	667	121,134	181.61	27.23

Table 4.34: *Number of vertices and edges of the modules networks of the Apache, GNOME and KDE projects.*

Project name	Commiters (Vertices)	Edges	Commiters per module	Avg Number of edges
Apache	751	23,324	4.3	31.06
KDE	915	205,877	12.5	225.00
GNOME	869	N/A	1.3	N/A

Table 4.35: *Number of vertices and edges of the commiters networks of the Apache and KDE projects.*

So, for instance, the Apache network of modules is composed of 175 modules with almost 2,500 edges interconnections. KDE has a lower number of modules as it groups many of its applications in suites (as the KOffice suite, that consists of the KWord word processor, the KSpread spreadsheet

application, the KPresenter presentation tool, and the Kivio flow-charting application, among others); in total, it is made up of 73 modules with over 1,500 relations among them. Finally, the number of vertices (CVS modules) for the GNOME project is 667 and the number of edges over 120,000. The high numbers for GNOME are because of a more anarchic structure of its CVS repository; modules are created for any application related to the GNOME project without many obstacles. Some modules do not even belong to the GNOME project, but contain applications that just use the same the tool-kit, GTK+, as GNOME.

As stated before, we will study the commiters networks only for Apache and KDE. In the case of Apache, we have identified 751 commiters in its CVS versioning system, and the number of relations among them is above 20,000. Regard that, in this case, connections in the commiters networks are given when two developers have worked on the same file. For KDE the number of commiters is similar (915 commiters), but the number of edges is an order of magnitude higher. The difference is due to the nature of both projects; while KDE is a monolithic project focused on a common goal working with a small set of technologies (the same Qt tool-kit, the same user interfaces, among others), the Apache project groups a wider set of projects. Projects in Apache have their specific goals, their specific technologies, etc. Hence, the lower barrier of entry for working on other modules in KDE is lower than the one for Apache.

By comparing the data in both tables some interesting conclusions can already be drawn. It may be observed, for instance, that the average number of commiters per module is greater in KDE (12.5) than in Apache (4.3), meaning more people being involved in the average KDE subproject. It can also be highlighted that the average degree on the commiters networks is in general larger than in the modules ones. This is specially true for KDE, which raises from a value of 21.4 in the latter case to 225 in the former. In the case of Apache it only raises from 14.2 to 31.1. Therefore, we can conclude that in those cases, commiters are much more linked than modules. The percentage of modules linked gives an idea of the synergy (in form of sharing information and experience) in a network as many modules have commiters in common. We can assume that this happens because of technical proximity between modules. Regarding our case studies, KDE and GNOME show percentages near 30%, while the average Apache module is only linked to 8% of the other modules in the versioning system. So, Apache is specially fragmented in several module families that have no commiters in common. KDE and GNOME have a higher cohesion, while there is more dispersion in Apache.

In the following we will study some specific aspects of all these networks, with the idea of illustrating both how the methodology is applied and which kind of results can be obtained from it.

4.6.5 Degree in the modules network

As we have already slightly discussed for Linux, the degree of vertices is one of the most relevant parameters to obtain information about the topology and evolution of a network. The most popular characterization of network degree is the distribution degree $P(k)$, which measures the probability of a given vertex having exactly k edges. However, the representation of $P(k)$ in networks of low size like ours is usually messy[15]. In this cases, the specialized literature prefers to use an associated parameter called the *cumulative distribution degree*, $CP(k)$, which is defined as:

$$CP(k) = \sum_{k}^{\infty} P(i) \tag{4.11}$$

The cumulative distribution degree is usually represented in a log-log scale.

Fig. 4.42 shows the cumulative distribution degree for our three networks. As it can be observed, all of them present a sharp cut-off, which is a symptom of an exponential fall of the distribution degree tail. From a practical point of view, this means that none of our networks follows a power law distribution. This is quite a remarkable finding, because the specialized literature has shown that most social networks present power laws for this parameter. This implies that the growth of the network does not follow the traditional random preferential attachment law. Thus, it is difficult to extract

[15]Social network analysis has been applied to networks with hundreds of thousands, sometimes millions of vertices. In this sense, our networks are of a small size even if we are handling large libre software projects.

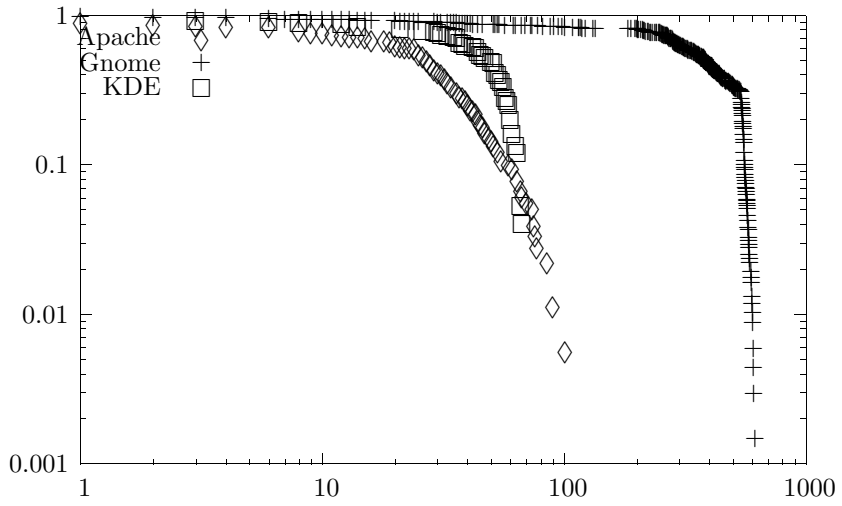


Figure 4.42: *Cumulative degree distribution for Apache (▽), KDE (+) and GNOME (·).*

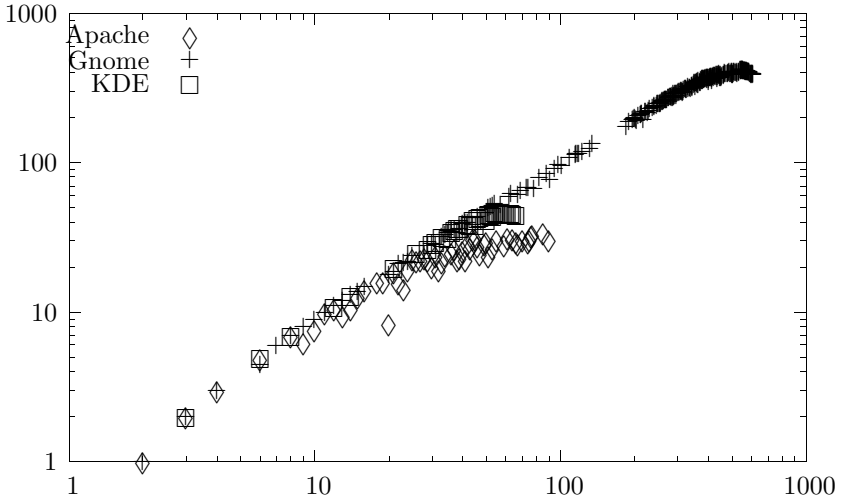


Figure 4.43: *Assortativity (degree - degree distribution) for Apache (▽), KDE (+) and GNOME (·).*

some conclusions at this point; maybe by using a weighted network approach, as we will see next, we can infer more information about the network topology.

Starting with the degree of the vertices we can also carry out an analysis of the assortativity of the networks. The assortativity measures the average degree of neighbors of vertices having a particular degree. For this reason we may also call it the *degree - degree distribution*.

Fig. 4.43 represents this parameter for our networks. As it can be observed, all three networks are elitist, in the sense that vertices tend to connect to other vertices having a similar degree ('rich' with 'rich' and 'poor' with 'poor'). This is specially clear in the case of GNOME, where the curve approaches to a linear equation with slope 1. Apache project deviates slightly from that behavior, showing some higher degree modules connected to other modules of a lower degree.

The previous analysis assumes unweighted networks. If we consider now weighted edges, we obtain similar conclusions. Fig. 4.44 represents the cumulative weighted distribution degree of the networks. Comparing this picture with fig. 4.42 we may remark that the sharp exponential cut-offs have disappeared. This is specially clear in the case of GNOME, where the curve tail can be clearly approximated by a power law. The interpretation for this finding is that the growth of that network could be driven by a preferential attachment law based on weighted degrees. This means that the probability of a new module to establish a link with a given vertex is proportional to the weighted

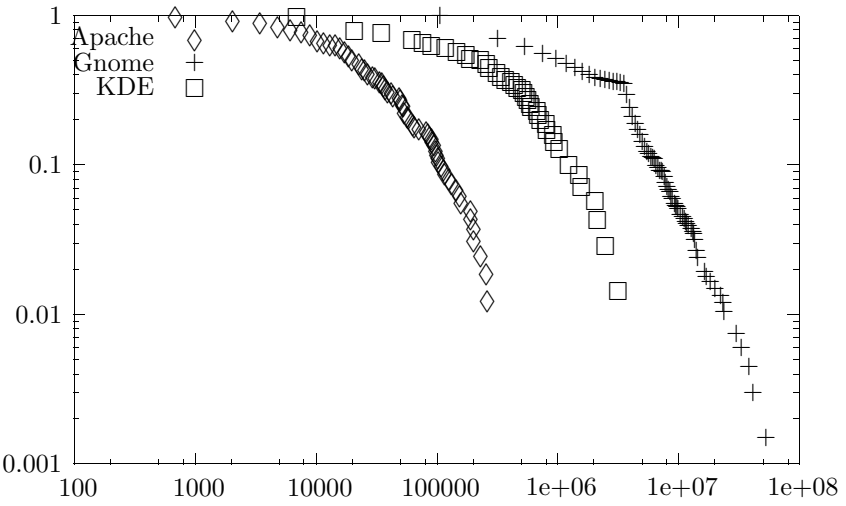


Figure 4.44: *Cumulative weighted degree distribution for Apache (▽), KDE (+) and GNOME (•).*

degree of that vertex. That is, the commiters of new modules are, with high probability, commiters of modules which are well connected (have high weighted degree) in the network. It should be noted that the use of weights has given a more realistic picture.

From figure 4.42 it can be remarked that the sharp cut-offs for Apache and KDE are close to each other. This means that the maximum number of relationships in both projects are similar. Nevertheless, observing fig. 4.44, it can be seen that the KDE tail is clearly over the Apache tail. This fact implies that KDE weighted links are, on average, stronger than those of Apache. We can quantitatively verify this by calculating the average edge weight for the three projects. If we do so, we obtain $1,409.27$ for Apache, $11,136.82$ for KDE and $7,661.18$ for GNOME.

If we multiply the average edge weight by the number of modules, the result will provide the total amount of commits performed by developers that contribute to at least two modules: $105,695$ for Apache, $812,988$ for KDE and $5,110,007$ for GNOME. This gives an idea of the modularity of the modules as a lower number of commits is indicative for developer work being more focused on a low number of modules. While the figures for Apache are not surprising (we have already noticed with previous parameters that there exist a high structuring of the Apache project), the difference between KDE and GNOME is astonishing. The organization of the KDE CVS repository yields in more independent modules than the ones found in the one for GNOME.

Without considering weights, we could observe in figure 4.42 that the sharp cut-offs for Apache and KDE were close to each other. This meant that the maximum number of relationships in both projects were similar. Nevertheless, observing figure 4.44 (now considering weighted links), the KDE tail is clearly over the Apache tail, what implies that KDE weighted links are, on average, stronger than those of Apache. This can quantitatively verified: we have calculated the average edge weight for the three projects obtaining $1,409.27$ for Apache, $11,136.82$ for KDE and $7,661.18$ for GNOME. Again GNOME and KDE show similar figures with KDE showing slightly higher vales (in this case meaning stronger links) while the numbers for Apache are very distant.

If we multiply the average edge weight by the number of modules, we obtain the total amount of commits performed by developers that contribute to at least two modules: $105,695$ for Apache, $812,988$ for KDE and $5,110,007$ for GNOME. This gives an idea of the modularity of the modules as a lower number of commits is indicative for developer work being more focused on a low number of modules. While the figures for Apache are not surprising (we have already noticed with previous parameters that there exist a high structuring of the Apache project), the difference between KDE and GNOME is astonishing. The organization of the KDE CVS repository yields in more independent modules than the ones found in the one for GNOME.

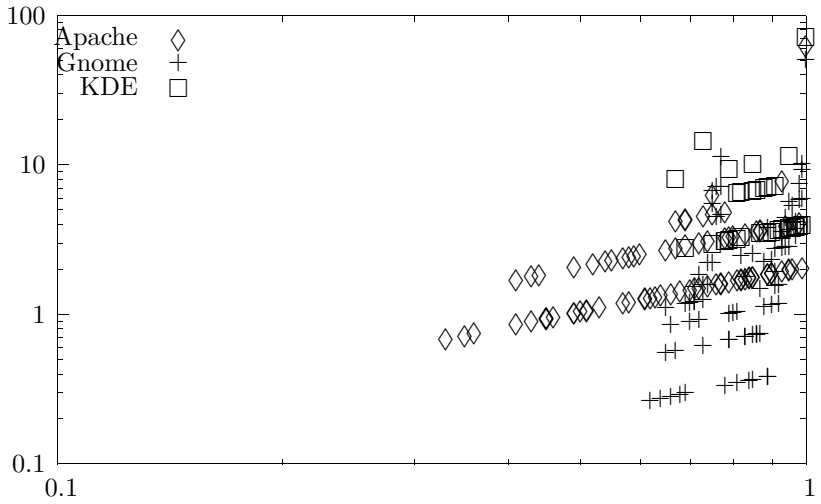


Figure 4.45: *Clustering coefficient distribution for Apache (▽), KDE (+) and GNOME (•).*

4.6.6 Clustering coefficient in the modules network

We have represented the distribution of the clustering coefficient in figure 4.45. From that figure we could observe how higher values of the clustering coefficient are the most typical for all the three networks. This is usually a symptom of *small world* behavior. The small world behavior of a network can be analyzed by comparing it with an equivalent (same number of vertices and edges) random network. When a network has a diameter (or average distance among vertices) similar to its random counterpart but, at the same time, has a higher average clustering coefficient, we say it is small world.

In table. 4.36 we represent the average distance $< d >$ among vertices and the average clustering coefficients $< cc >$ for our three networks and their equivalent random counterparts ($< rd >$ is the random average distance and $< rcc >$ is the random average clustering coefficient). As it can be observed, the three networks satisfy the small world condition, since their average distances are slightly above to those of their random counterparts; but the clustering coefficients are clearly higher.

Project name	Avg distance/Random avg distance ($< d > / < rd >$)	Avg clustering coef/Random avg clustering coef ($< cc > / < rcc >$)
Apache	2.06 / 1.47	0.73 / 0.19
KDE	1.31 / 1.11	0.88 / 0.65
GNOME	1.46 / 1.10	0.87 / 0.54

Table 4.36: *Small world analysis for the modules networks. The second column contains the average distance and the average random distance. The third column gives the average clustering coefficients and the random average clustering coefficients.*

The average random clustering coefficients for KDE and GNOME are very close to the real ones, due to the high density of those networks. This could be an indication of over-redundancy in their links. That would mean that the same efficiency of information could be obtained with less relationships (i.e. we could eliminate many edges in the network without significantly increasing the diameter or reducing the clustering coefficient). In this sense, the Apache network seems to be more *optimized*. To interpret this fact, the reader may remember that links in this network are related to the existence of common developers for the linked modules. It should be noted that redundancy is probably a good characteristic of a libre software project as it may lose many developers without being affected heavily. It may be especially interesting to have over-redundancy in projects with many volunteers, as in those environments turn-over may be high. Future research should focus on investigation if over-redundancy is a good or bad parameter in the case of libre software projects. On the other side, how much of this redundancy is due to have taken a static picture of the project should be researched; it may well be

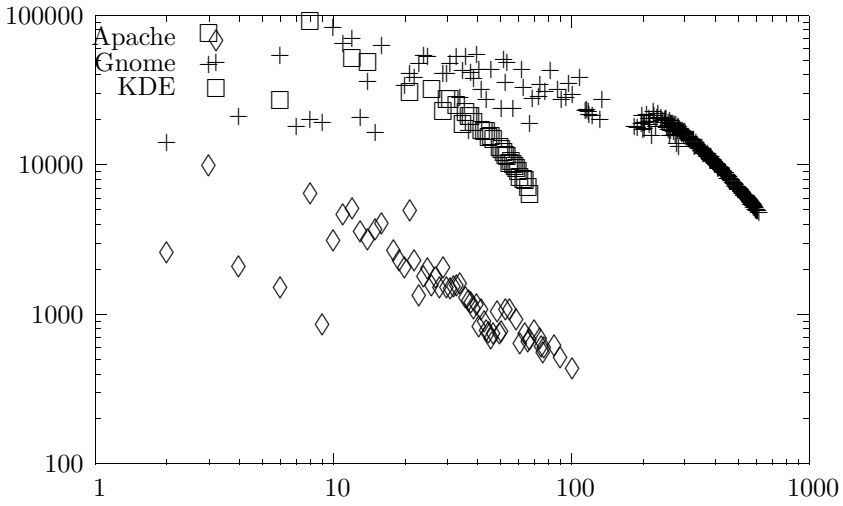


Figure 4.46: *Average weighted clustering coefficient as a function of the degree of vertices for Apache (▽), KDE (+) and GNOME (•).*

that the redundancy we have observed is the result of different generations of developers working on the same file in different periods of time.

Some interesting conclusions can also be obtained by looking at the weighted clustering coefficient. In fig. 4.46 we can observe the average weighted clustering coefficient as a function of the degree of vertices (the weighted degree - degree distribution). As we have already noticed, the KDE and GNOME networks have a similar local redundancy, which is higher than the one of Apache. High redundancy implies more fluid information exchanges in the short distances for the first two projects. Besides, the weighted clustering coefficient in all cases falls with the degree, according to a power law function. We can infer that highly connected vertices cannot maintain their neighbors as closely related as poorly connected ones. This happens typically in most social networks because the cost of maintaining close relationships in small groups is much lower than the equivalent cost for large neighborhoods.

It is well known that small world networks are those optimizing the short and long term information flow efficiency [Watts, 2003]. Those networks are also especially well adapted to solve the problem of searching knowledge through their vertices. As we can observe, the average random clustering coefficient for KDE and GNOME are very close to the random one due to the high density of those networks. This could be an indication of over-redundancy in their links which could mean that the same efficiency of information could be obtained with less relationships (i.e. we could eliminate many edges in the network without significantly increasing the diameter or reducing the clustering coefficient). In this sense, the Apache network seems to be more optimized. To interpret this fact, the reader may remember that links in this network are related to the existence of common developers for the linked modules. Having two or more developers in common means then having redundancy; the positive aspect of redundancy is, of course, that if one of the developer quits the project the link still persists; hence, redundancy makes the network more resilient.

4.6.7 Distance centrality in the modules network

The analysis of the distance centrality of vertices is relevant because this parameter measures how close a vertex is to the rest of the network. In fig. 4.47 we can observe the distance centrality distribution for our three networks. They follow multiple power laws, making higher values of the parameter most probable. This is an indication of well structured networks for the fast spread of knowledge and information.

We can also analyze the average distance centrality as a function of the degree (average distance centrality - degree distribution), which is shown if Fig. 4.48. It can be observed that in all three cases the average distance centrality grows with the degree following, approximately, a power law of

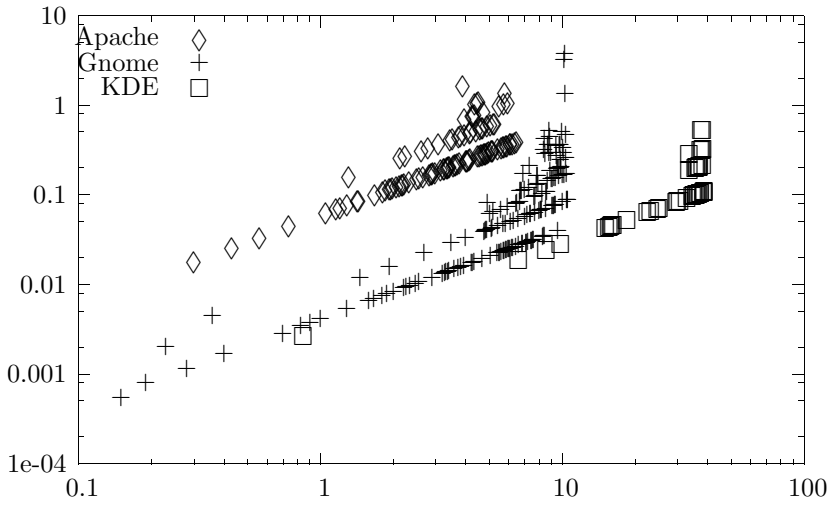


Figure 4.47: *Distance centrality distribution for Apache (▽), KDE (+) and GNOME (•).*

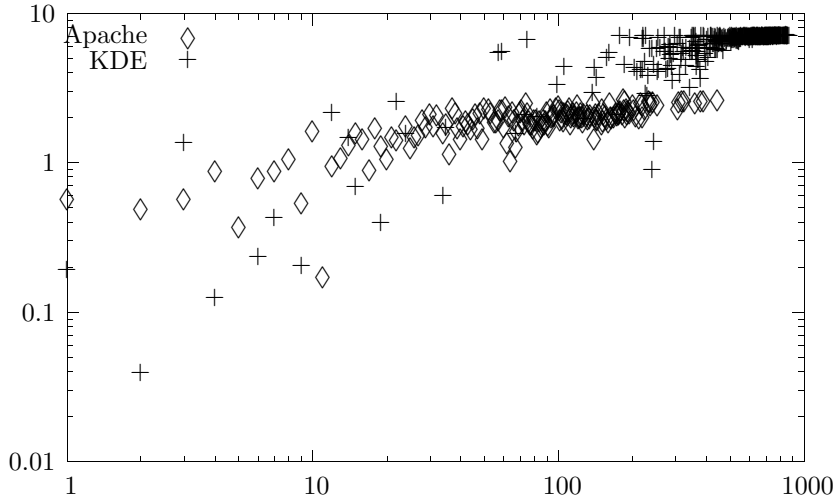


Figure 4.48: *Average distance centrality as a function of the degree of vertices for Apache (▽), KDE (+) and GNOME (•).*

low exponent. This means that, in terms of distance centrality, the networks are quite *democratic*, because there is not a clear advantage of well connected nodes respect to the rest. Curiously enough, the Apache and GNOME curves are quite similar, while the KDE one is clearly an order of magnitude over the rest. This could be an effect of the lower size of this network, but is also an indication of a specially well structured network in terms of information spread. So, even if KDE showed to be more modular as we have seen for a previous parameter, its structure seems to maximize information flow.

4.6.8 Betweenness centrality in the modules network

The distance centrality of a vertex indicates how well new knowledge created in a vertex spreads to the rest of the network. On the other hand, betweenness centrality is a measurement of how easy it is for a vertex to generate this new information. Vertices with high betweenness centrality indexes are the crossroads of organizations, where information from different origins can be intercepted, analyzed or manipulated. In fig. 4.49 we can observe the betweenness centrality distribution for our three networks. As it was the case of the distance centrality, it grows following a multiple power law. Nevertheless, there is a significant difference between the distribution of these two parameters. Although the log-log scale of the axis of fig. 4.49 does not allow to visualize it, the most probable value of the betweenness

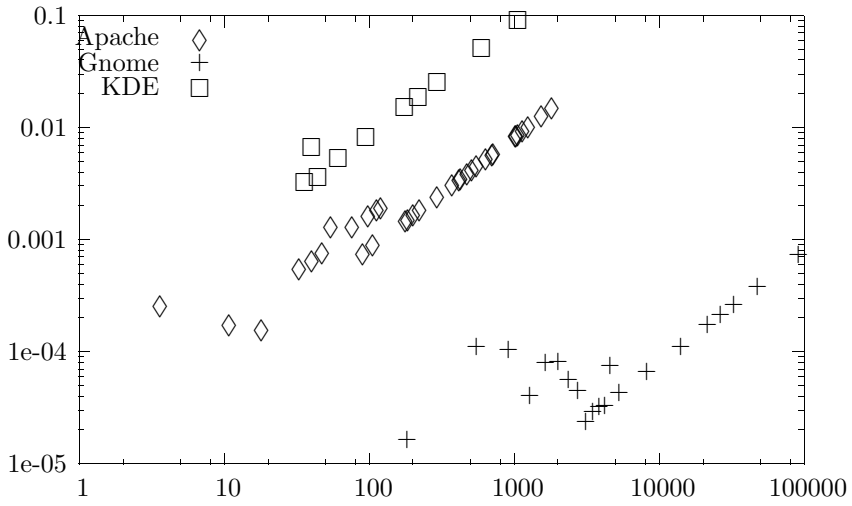


Figure 4.49: *Betweenness centrality distribution for Apache (▽), KDE (+) and GNOME (·).*

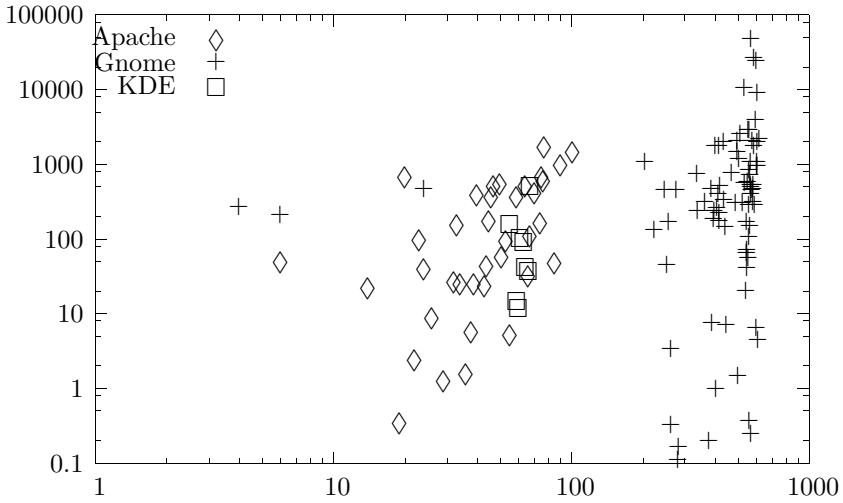


Figure 4.50: *Average betweenness centrality distribution for Apache (▽), KDE (+) and GNOME (·).*

centrality in all three networks is zero. Just to show an example, only 102 out of 677 vertices of the GNOME network have a nonzero betweenness centrality. So, the distance centrality is a common good of all members of the network, while the betweenness centrality is owned by a reduced elite. This should not be surprising at all, as projects usually have modules (i.e. applications) which have a more central position and attract more development attention. Surrounding these modules other, minor modules may appear.

This fact can also be visualized in fig. 4.50, where we represent the average betweenness centrality as a function of the degree. It can be clearly seen that only vertices of high degree have nonzero betweenness centralities.

4.6.9 Commiters networks

The analysis of commiters networks draw similar conclusions to those shown for modules networks, and therefore they are not going to be commented in detail. For instance, the cumulative degree distribution for the two commiters networks is shown in fig. 4.51, which has clearly the same qualitative properties than for this parameter for the modules networks shown in fig. 4.42. The same holds for the commiter cumulative weighted degree distribution depicted in fig. 4.52, or for the average degree as a function of degree, depicted in fig. 4.53, where it can be noticed how both networks maintain the

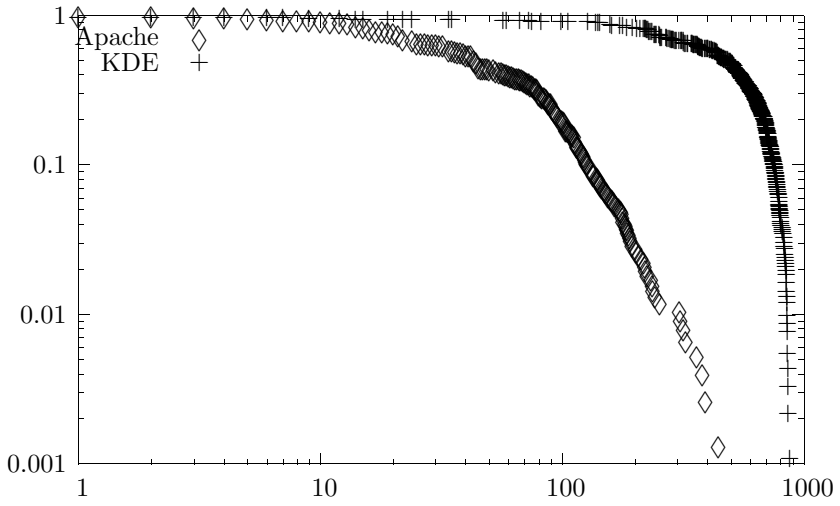


Figure 4.51: *Cumulative degree distribution for Apache (▽) and KDE (+).*

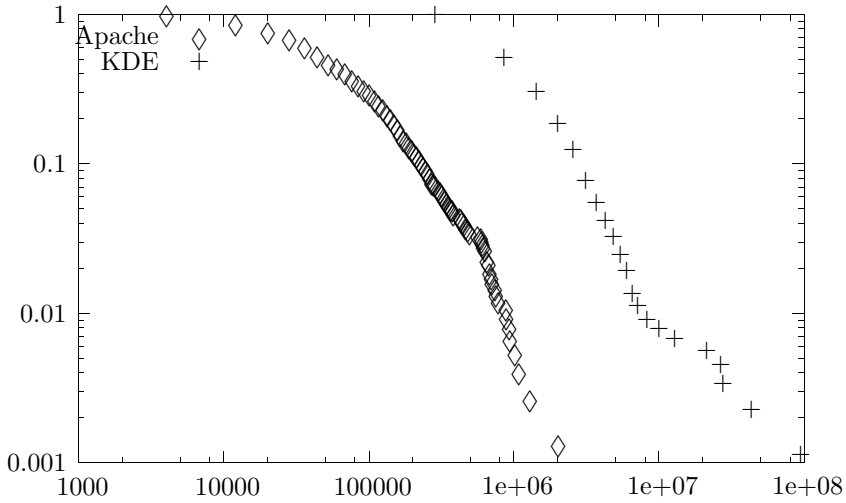


Figure 4.52: *Cumulative weighted degree distribution for Apache (▽) and KDE (+).*

elitist characteristic observed also in the case of modules.

An interesting feature of commiters networks can be seen in fig. 4.54. The average weighted degree of authors remains more or less constant for low values of the degree. Nevertheless, in the case of KDE, it increases meaningfully for the highest degrees. The implication of this is that commiters with higher degrees not only have more relationships than the rest, but also their relationships are much stronger than the average. This indicates that authors having higher degrees are more involved in the project development and establish stronger links than the rest. At the same time, as we observed in fig. 4.53, they only relate to other commiters being compelled in the project in their same degrees. If this behavior is found in other large libre software projects, it could be a valid method to identify the leading *core group* of a libre software project. On the other hand, the Apache project seems to promote a single category of developers, given that the weighted degree does not depend so clearly on the degree of vertices. It may be also that because of the fragmentation of the Apache project in many *families* of modules, it is easy to developers to reach a point where they do not have the possibility to get many more developers known.

Table 4.37 digs into the small-world properties of commiters networks. As we can observe, both networks can still considered to be small world. The Apache case is specially interesting, because an increase in the average distance is observed. This characteristic plus the large value of the clustering

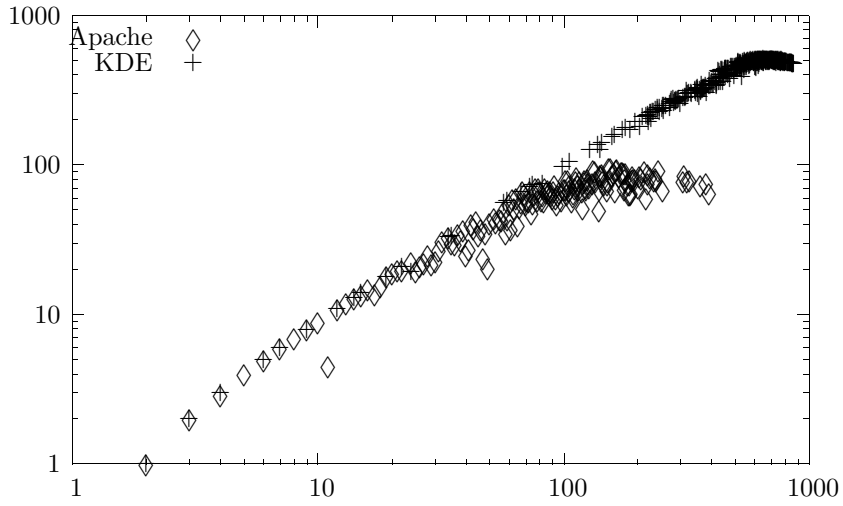


Figure 4.53: *Degree - degree distribution for Apache (▽) and KDE (+).*

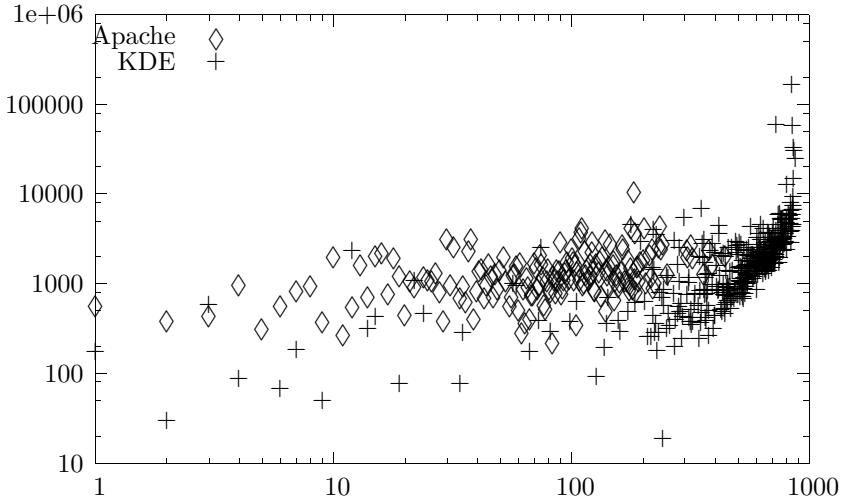


Figure 4.54: *Average weighted degree as a function of the degree for Apache (▽) and KDE (+).*

coefficient may indicate that the network is forming *cliques*.

Project name	Avg distance/Random avg distance ($< d > / < rd >$)	Avg clustering coef/Random avg clustering coef ($< cc > / < rcc >$)
Apache	2.18 / 1.60	0.84 / 0.08
KDE	1.47 / 1.10	0.86 / 0.52

Table 4.37: *Small-world analysis for commiters networks.*

4.6.10 Inferring the social structure of a project

A *classical* network analysis of a large libre software project would offer a graph as in figure 4.55, where the social structure could be hardly inferred. The small size of the Linux 1.0 kernel version has been the reason why we could analyze it visually (see subsection 4.6.3), but not later, larger versions. Other techniques have to be used for these larger projects such as GNOME, Apache or KDE (see 4.6.10).

In this subsection we are going to apply an additional technique to data obtained from the CVS versioning repositories. This method will allow to visually infer the community structure of a software

project by means of the Girvan-Newman (GN)algorithm [Girvan & Newman, 2002]. The GN algorithm generates a binary tree optimizing the layout so that branches, that represent real communities, are as clear as possible.

Regarding our methodology, we believe that this technique can be used to obtain and compare the community structure of large libre software projects. From the structure, researchers may infer whether there are strong communities within projects or not. Introducing longitudinal data opens the possibility to study the evolution of these communities, and how their interrelationships changes. The observations on these studies can be used to estimate information flow within the project, collaboration patterns, and other important characteristics of projects.

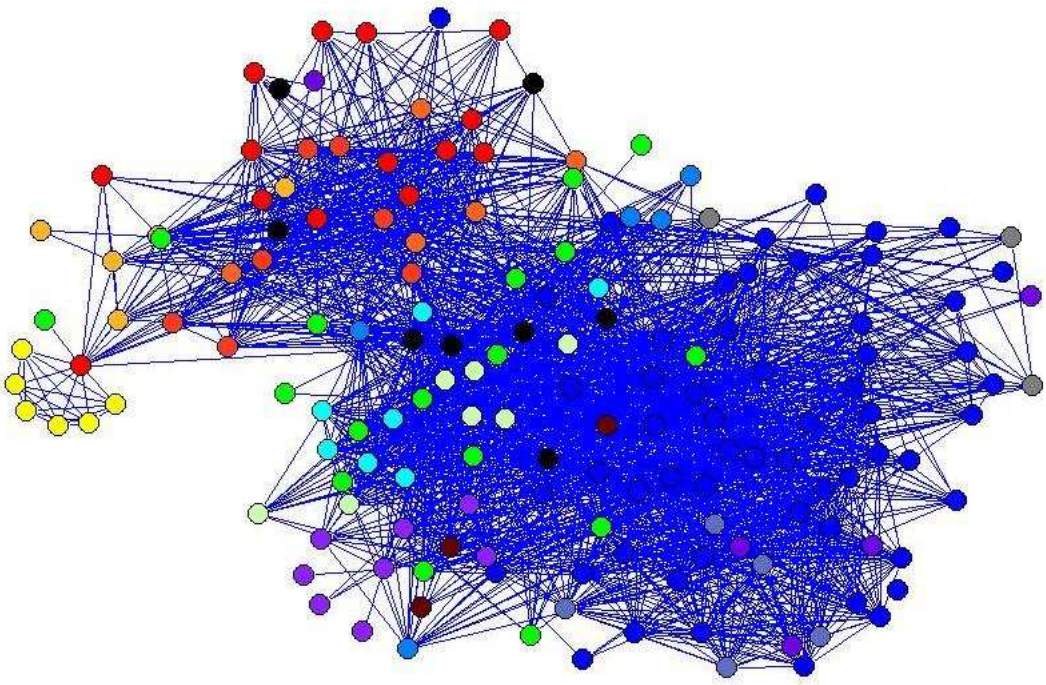


Figure 4.55: Classical *network analysis on the Apache modules for February 1st, 2004.*

Evolution of the structure of the modules network in Apache

We have applied the GN algorithm to the modules of the Apache project. To color the nodes, we follow in general the naming conventions of the project, which correspond to *families* of modules. That way, we have used red for those modules related to the HTTP server (apache/httpd), orange for those related to modperl, green for XML related modules, dark blue for Jakarta, light blue for Avalon, yellow for TCL-related modules and black for generic modules like CVSROOT (administrative files of the CVS repository) or the web site (site). A complete list of all families and colors can be found in table 4.38.

On the other hand, the radius of the vertices changes from module to module as is depends on the software size (measured in LOC). So, small vertices are indicative for small modules, while big vertices represent modules with many lines. It should be noted, that we are not considering source lines of code, so all type of lines are taken into account. In this sense, the XML-related modules (green) will appear very large in size, although most probably their code base is small.

Thanks to the branches and the colors, we can observe the structure of the project. For instance, in January 1999 when the number of modules in the Apache repository was around a dozen (shown in figure 4.56), modules related to modperl are seen only in the upper-left branch. This means that those modules already had their own developer community working on them.

The Apache project grew significantly from 1999 onwards. Figure 4.57 depicts the community structure for January 2000, just 12 months after the previous figure. In this case, we can clearly

Module family	Color
httpd	red
jakarta	blue
xml	green
cocoon	light green
avalon	cyan
apache	red orange
tcl	yellow
perl	dandelion
mave	cadet blue
apr	orange
incubator	royal blue
logging	gray
db-	fuchsia
ws-	purple
ant	brown

Table 4.38: *Colors used for the different module* families *of the Apache project.*

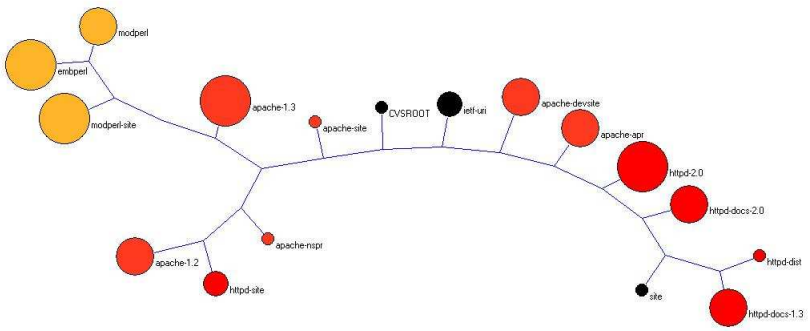


Figure 4.56: *Community structure of the Apache modules on January 1st, 1999.*

identify the modperl modules in a branch (in the center of the figure in orange) and a branch for Jakarta (blue) and XML-related (green) modules. The modules that correspond to the Apache HTTP server are those in red. They are situated on upper right-side branch.

Just nine months later, in September 2000 (see figure 4.58), Jakarta has differentiated itself (we can observe two close branches at the bottom of the figure) from XML-related modules (branch at the bottom right). On the other hand, modperl modules are now spread through several branches, mixed with the apache/httpd modules (top part of the figure). Noteworthy is the fact that the number of modules has almost doubled in just nine months. This growth is especially significant for the Jakarta modules; not only in terms of number of modules, but also regarding the size of the modules as we can see from the radius of the vertices that correspond to Jakarta modules (in blue).

From the figures for early 2002 (figure 4.59 and early 2004 (figure 4.60) we can infer that the community structure of the Apache modules has reached its maturity. The number of modules has grown in the interval that goes from 2002 to 2004, but the structure of the network has not changed much. New in the year 2002 are the Avalon (in light blue) and TCL-related (in yellow) modules that appear both clearly as separate branches in the center of the figure. For the rest of modules, the XML modules (green) are mostly situated inside the Jakarta (blue) modules, while the modperl (orange)

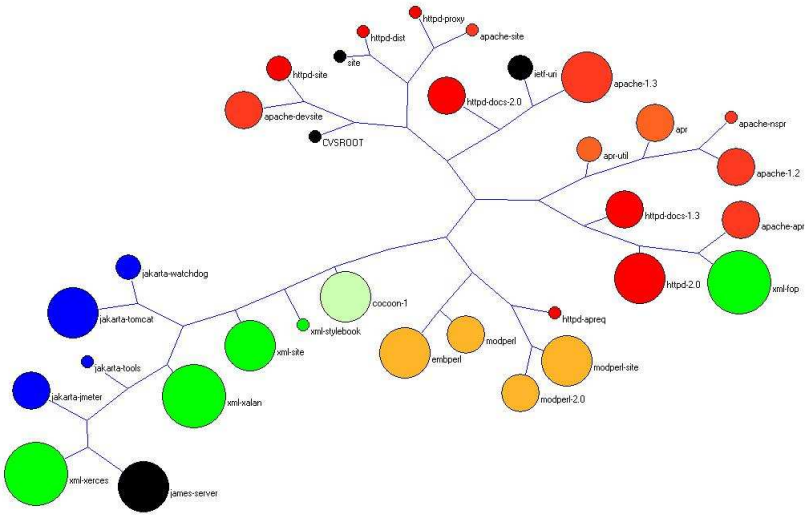


Figure 4.57: *Community structure of the Apache modules on January 1st, 2000*

modules seem to be mixed with the httpd (red) ones.

In the final figure, taken for February 2004, modperl (orange) modules while still being close to the httpd (red) ones, are now more independent and are situated mostly in their own branch. Avalon (light blue), which two years later could be clearly be differentiated as a unique community, is now split in three branches and other modules, mainly related to Jakarta and XML, can also be found. Jakarta (dark blue) has grown to the point that it supposes the largest part of the Apache project and is present in several communities, usually intermixed with XML (green) modules.

In summary, from the study of the evolution of the community structure of the Apache modules, we can conclude that the Apache project has a first phase (from 1999 to 2002) where its structure was not defined. Around 2002 the Apache project reached an almost-mature status and although the number of modules kept on growing, its structure has remained almost the same. Hence, even if the project grows a lot during the last phase, the conservation of the structure means that the pairs of nodes related by the developers working on them remain more or less the same. In addition, we con infer that developers tend to work in the projects within their *thematic community*. But even within these communities, clear differences exist: Jakarta-XML modules have a tendency to build large branches, while other *thematic communities* seem to be smaller. Another finding is that apache/httpd-related modules (in red) and Jakarta-related modules represent separate communities for all dates considered. Needless to say technological aspects are a key fact for understanding this (Jakarta is mainly based in Java, while apache/httpd uses basically C), but a in-depth study of both communities may also provide further differences.

Another conclusion that we can draw from our analysis is that in spite of the findings by Mockus et al. [Mockus *et al.*, 2002] we have not found an effective splitting of modules when the size of the main contributing group exceeds a certain number of persons. In fact, two other behaviors, similar in their effect, can be deduced: the emergence of new modules around consolidated ones and the specialization of developers (which can be can be derived by the fact that a branch splits in two).

Structure of the Apache and the GNOME developer networks

So far we have discussed the structure of the CVS modules the Apache project and how this structure evolves. Next, we are going to apply this technique to the developers networks. In this case we are not going to consider how this network evolves and will focus only in one point in time: February 2004. Figure 4.61 displays the binary tree obtained after applying the GN algorithm to the developers network of Apache and GNOME.

Each node corresponds to a commiter who has done at least a commit to the CVS versioning

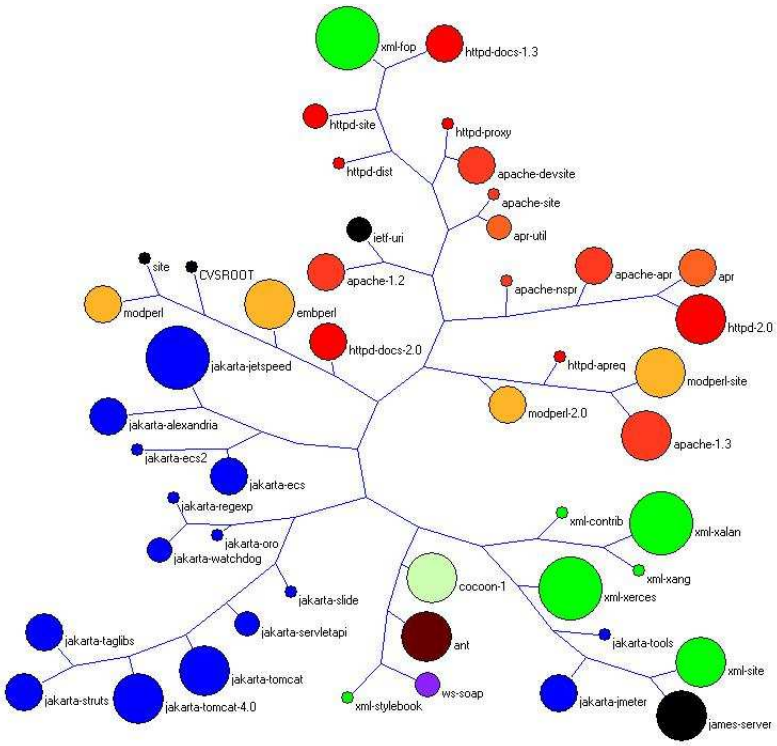


Figure 4.58: *Community structure of the Apache modules on September 1st, 2000*

repository. Developers (nodes) of the Apache project are colored by the subprojects they have most contributed to as we have done in the previous analysis on modules (see table 4.38 for the distribution of colors). The results we have obtained with the developer network are similar than the ones for modules. We can see two clearly differentiated branches, one on the left which is mostly due to the apache/httpd modules (in red) and one that is vertical that corresponds to modules of the Jakarta family (in blue). In between we can see that there exist clusters of all the other families we have defined. With the exception of Avalon (light blue) which is spread along the Jakarta modules, all other *module families* can be clearly localized in branches on their own. This means that we may define several communities in the Apache project and that these communities are, as modules were, very dependent on technological issues.

With some exceptions for the upper tail of the figure, the structure for the GNOME project is similar to the binary tree obtained from a random graph. So, we cannot infer any structure for GNOME using algorithm. This makes sense with previous findings of the social network analysis on modules. The GNOME project, being technologically less dispersed (GNOME makes use of a common graphical tool-kit (GTK+) and a set of shared libraries) than the Apache project, has many more ties among developers.

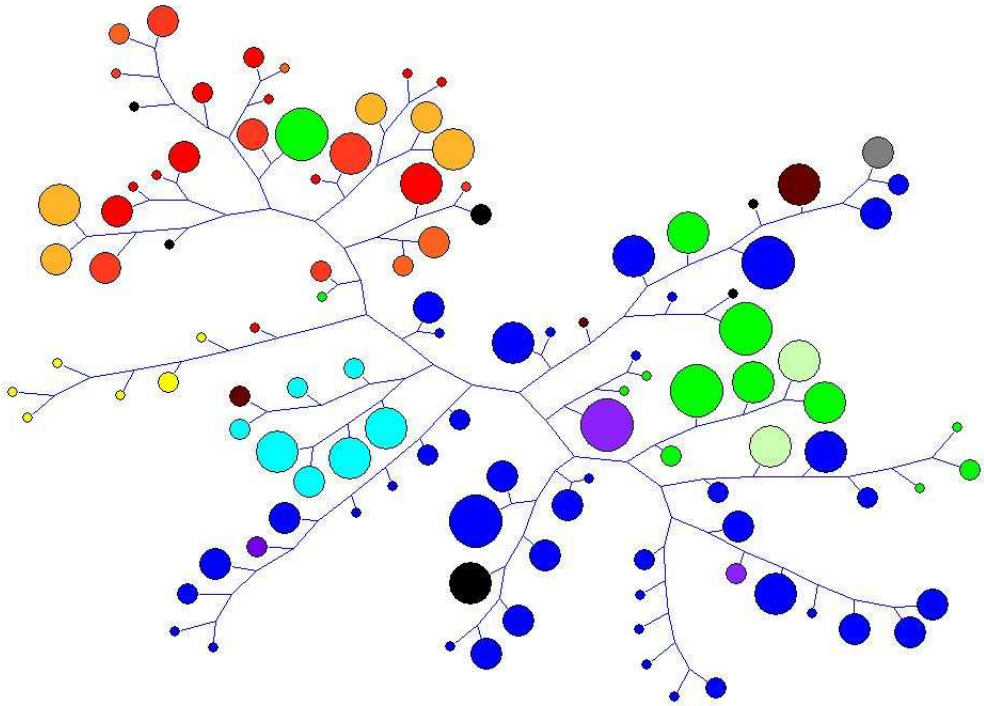


Figure 4.59: *Community structure of the Apache modules on January 1st, 2002*

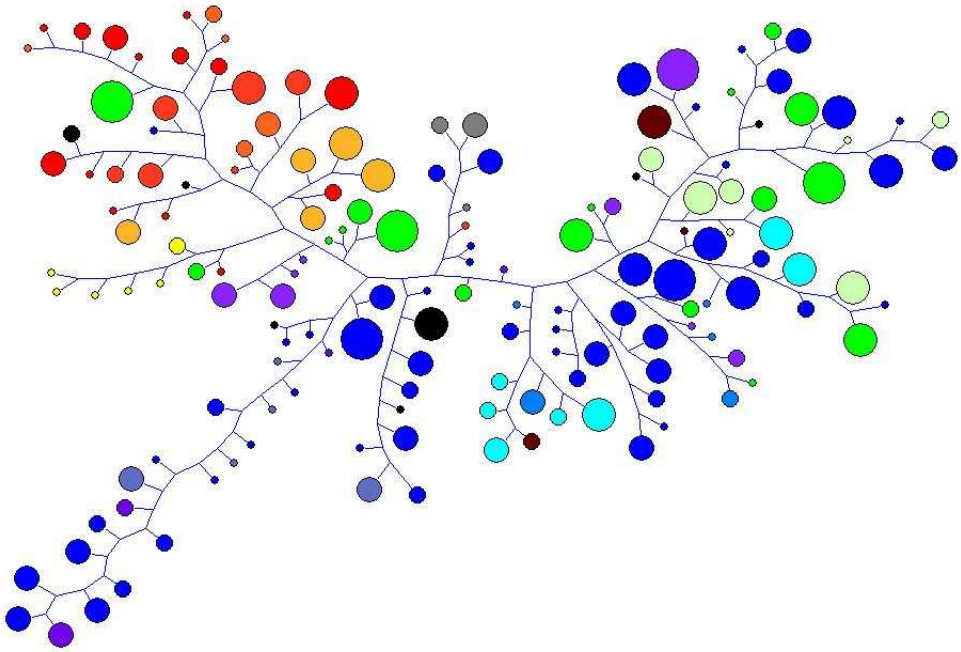


Figure 4.60: *Community structure of the Apache modules on February 1st, 2004*

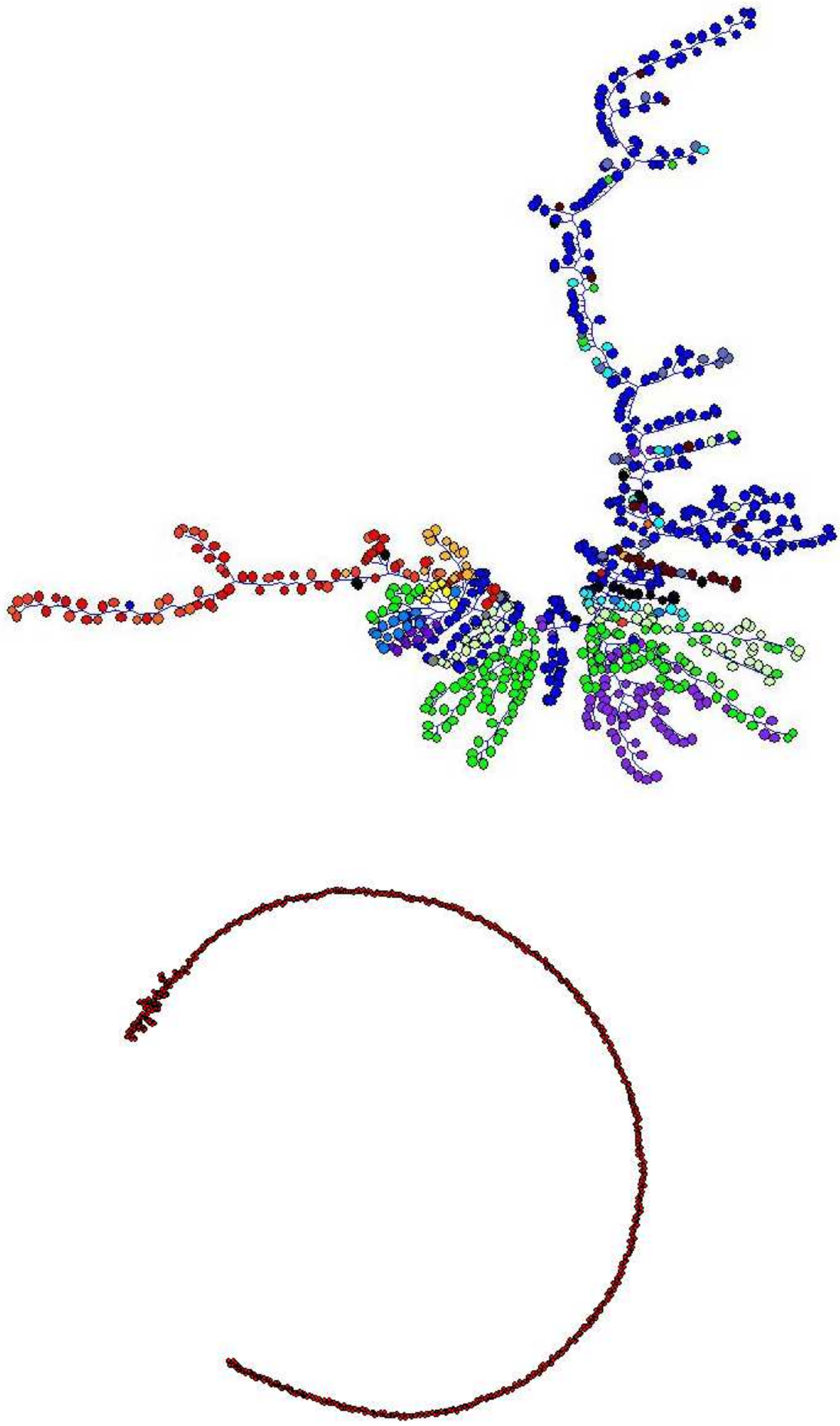


Figure 4.61: *The first picture represents the community structure of the Apache commiters network obtained by the application of the GN algorithm (February 2004). The picture at the bottom depicts the community structure of the GNOME project (February 2004).*

.

4.7 Evolution of contributors

Libre software development is heavily based on voluntary contributions. Developers are self-selected, i.e. they choose the project(s) they want to contribute to, and they can abandon their activity anytime. But, as libre software projects evolve, changes in the development team affect their organization and decision structure. New members enter the group and others leave. From the related research we know that a small group of very active developers is responsible in general for the proper evolution of a project (see subsection 2.4.1). But this view only gives a static picture. It does not attend to the time axis that evolution requires. In this section, we want to focus on this issue and analyze how software developers evolve in large libre software projects. As case studies, we have selected Debian and a set of libre software projects.

Debian has been selected as all its contributors are volunteers and thus offers a good choice for researching the behaviour of such kind of contributors without interference from developers who are full or part-time hired by a company to work on a libre software project. Given the importance and volume of the Debian project, it is worth studying the evolution of the people maintaining packages (also known as maintainers). Our aim is to give quantitative insight about the evolution of Debian maintainers in the last six and a half years (from July 1998 to December 2004). We study how many of these developers remain from the beginning of the interval (from July 1998) and what happened to packages maintained by those developers who left the project. It should be noted that of the many tasks performed in the Debian project (ranging from the administration of the infrastructure to the translation of help texts and documentation into a number of supported languages), only packaging and maintenance activities are taken into account in this investigation. Packaging and maintenance are the more notorious activities and most contributors in Debian are devoted to them. They are also the most easy to quantify.

The other projects are libre software applications from GNOME, KDE, and FreeBSD. They are large in size in terms of source lines of code and of contributors, so we can assume that they are bazaar-driven projects. Specifically, we are interested in the *core group* of these applications to study how its composition changes over time.

4.7.1 Goals

The main goal of the study in this section is to analyze the evolution of the number of contributors in libre software projects and the activities these contributors perform. We will focus first on the number of contributors and analyse if the output they achieve (in terms of packages per contributor in the case of Debian) remains constant over time.

Then, we will determine how many maintainers remain active in subsequent releases. We want to measure the *volatility* of volunteers in a libre software project. That is, do developers join the project and work on it for short periods of time, or on the contrary do they stay for many years? Specifically, we have calculated the half-life of contributors in the project (half-life being the time required for a certain population of maintainers to fall to half of its initial size). This quantity could be easily compared with other libre software projects and with statistics from companies from the software and other industries.

Another question we want to target is how the contributions of maintainers evolve. Giving answer to this issue will allow us to know if *older* maintainers strengthen their contributions, maintaining more packages, as time passes. We expect exactly this result, because it seems reasonable that more experience leads to more efficiency in the maintenance of packages. Hence, more experience should be reflected in an increase in the number of maintained packages.

Up to now, our questions are related to those maintainers who stay in the project. From a different perspective it is interesting to research what happens to those tasks (i.e. packages) assigned to maintainers who have left the project. Since they are volunteers, maintainers may leave the project almost anytime. As a result, their packages become unmaintained. There are two chances for these packages: they may be taken over (adopted) by other maintainers, or they may be excluded from future stable releases. Exclusion from future releases happens to packages that do not fulfill the requirements of the Debian policy and the quality standards of the project and for unmaintained

packages the probability of falling behind in quality is very high. Our intention is to know how the abandonment of voluntary contributors affects Debian, and how this is damped down by other (possibly new) maintainers. In a sense this question targets how well Debian regenerates itself and survives the loss of some of its human resources.

The last question we want to answer for Debian is if maintainers who have stayed longer in the project are responsible for packages that are most commonly used by users. It seems reasonable to think that more experienced maintainers are in charge of more central packages as probably most used packages have been introduced in earlier releases.

Finally, we want to gain more knowledge on the evolution of the *core group* of a libre software project. This issue is an important one as a vast amount of work is done by this reduced *elite* and the project's health depends heavily on them. On the other hand, the abandonment of a *core group* developer cannot be equally evaluated as if any other casual contributor leaves the project. That is the reason why with the help of over a dozen case studies, we study the evolution of the *core group* of some large libre software projects. We have selected different case studies than for the previous research questions, as Debian is really an integration project and not a development one.ç Although it is possible that Debian counts also on a *core group*, previous research has been always centered on software development projects. In any case, our goal for these analyses will be to find out how the composition of the *core group* changes over the years. We therefore propose two scenarios: one where the composition has remained almost with any change (we will call it *code god*, as this group seems to possess an unachievable position to others) and another one where multiple generations can be observed. An intermediate scenario is also possible.

4.7.2 Methodology

The first set of research goals will be answered by analyzing publicly available information from the Debian project. So, our data sources are the Sources.gz (described in subsection 3.6.1) and the Debian Popularity Contest (presented in subsection 3.6.2).

While the acquisition of data is straightforward because of its public nature, there are some tasks that had to be performed to ensure the correctness of our results. One of the main problems we face while parsing the contents of the Sources.gz file is the identification and merge of different entries for the same maintainer. This occurs due to changes in the e-mail address or because of several spellings of the name (the middle name or the nickname are sometimes present while others it is left out, among others). We also have to resolve some inconsistencies between the information in the Sources.gz file and the data from the Debian Popularity Contest. While the former contains entries for source packages (platform-independent packages that contain the source code of the applications), the latter tracks statistics for binary packages (platform-dependent packages that include the output of the compilation process). Fortunately, we can link source packages to binary packages using the information in Sources.gz, as it includes for every source package the binary packages available in pre-compiled form.

The second type of analysis, the one concerned with the evolution of the core group, will be based on data obtained from CVS versioning repositories, more precisely the log history of these repositories. For the purpose of analyzing this information, we split the duration of the project in several (in our study, ten) intervals of the same duration. For each interval, we identify the 20% of developers (rounded by excess) who have contributed with more commits during that interval. This group is what we define as the *core group* during that interval. We then study the evolution of the commits made by the different *core groups* during the whole history of the project. It is important to notice that a given *core group* can have developers in common with the *core group* of any other interval (in general, they will, and in some cases, the core for several intervals can be exactly the same). The case studies will serve as a good example to clarify the process.

4.7.3 Evolution of the number of Debian maintainers

The information of the evolution in number of Debian maintainers provides us some basic data that will be useful in subsequent subsections. When we started the study, we expected a steady increase

of maintainers over time, as we knew the number of packages in Debian has been growing [González-Barahona *et al.*, 2004]. In fact, we anticipated the packages to maintainer ratio to be nearly constant, since it seems reasonable to consider that volunteers devote similar amounts of effort over time.

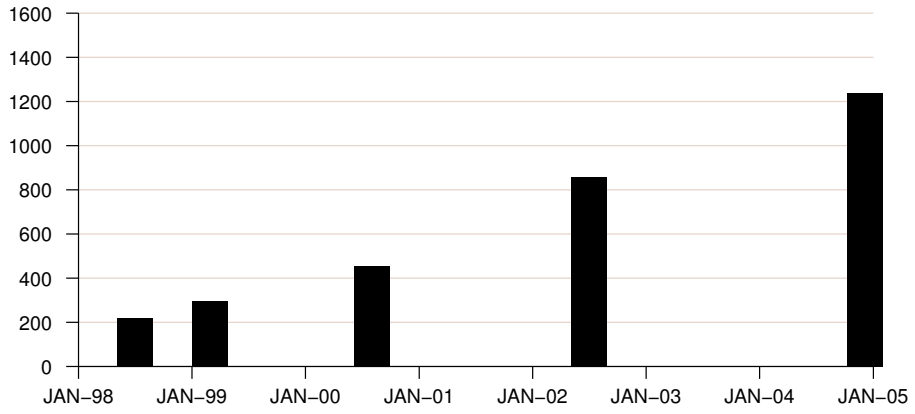


Figure 4.62: *Number of maintainers over time*

Figure 4.62 shows the evolution of the number of Debian maintainers for the latest five stable releases. As we assumed, the number of Debian individual maintainers has been growing. Debian 2.0 (July 1998) was put together by 216 individual maintainers, while the number of maintainers for later releases are 859 for Debian 3.0 (July 2002) and is of about 1,237 for the still unstable Debian 3.1 version (December 2004)[16]. This yields a growth rate of about 35% every year.

Date	Release	Maint	Packages	Pkg/Maint	Median	Mode	Std. Dev	Gini	Max
Jul 98	2.0	216	1,101	5.1	3	1 (52)	5.8	0.492	50
Mar 99	2.1	296	1,559	5.3	3	1 (76)	6.5	0.521	55
Aug 00	2.2	453	2,601	5.7	3	1 (122)	7.4	0.535	69
Jul 02	3.0	859	5,119	6.0	4	1 (208)	8.2	0.539	79
Dec 04	(3.1)	1237	7,786	6.3	3	1 (348)	9.5	0.574	121

Table 4.39: *Statistical analysis of the growth in number of Debian maintainers. First column gives the date of the release specified in the second one. "Maint" is the number of maintainers that maintain at least a package, "Packages" the number of total packages for that release, "Pkg/Maint" the mean number of packages per maintainer, "Median" the median number of packages, "Mode" gives the most frequent contribution in number of packages and in brackets the number of maintainers who contribute it, "Std. Dev" the standard deviation of our sample", "Gini" the Gini coefficient and "Max" the maximum number of packages that a unique maintainer is responsible for.*

We have performed a small statistical analysis of the data; results are shown in table 4.39. From this table we can observe that, contrary to our initial hypothesis, the ratio of packages per maintainer (see column "Pkg/Maint") does not remain constant and grows over time. The growth of packages is actually bigger than that of volunteers contributing. Some possible explanations for this finding may be found in the introduction of new development and maintenance tools or in newer, more efficient practices. Devoting the same amount of time to Debian would possibly mean maintaining more packages.

Noteworthy is that the median does not vary (with the exception of Debian 3.0) in these last years. Half of the maintainer population does not have more than three packages to maintain. The mode, on the other hand, shows that the most frequent situation is a maintainer maintaining one package.

[16] At the time of writing this section, the next stable release, 3.1, was still in preparation. This is why '3.1' will appear in parenthesis in some tables. The Debian 3.1 version was finally released in June 2005. We believe that the data included in this section for the candidate are very near to those of the final version.

In brackets we can find the number of developers who actually maintain only one package which is around one forth of the total population of Debian maintainers.

The next three values (the standard deviation, the Gini coefficient and the maximum number of packages maintained by a single maintainer) strengthen the idea that the distribution of work tends to be in a more unequal way. This means that a small number of maintainers is in charge of an increasing number of packages while the number of packages the vast majority maintains does not change much. Compared to other libre software applications and in general to other studies which have looked at the distribution of work in libre software projects [Ghosh & Prakash, 2000; Koch & Schneider, 2002; Hunt & Johnson, 2002; Mockus *et al.*, 2002; Ghosh *et al.*, 2002a], we can see that the Debian project is far away from a Pareto distribution and from values of the Gini coefficient found in the activity of CVS repositories that range between 0.7 and 0.9. In other words, a reduced *elite* that is responsible for a vast amount of work does not exist. This may be explained from the technical *nature* of Debian: it is a project that integrates software from third parties and not a project with software development as its primary goals. Nevertheless, the Gini coefficient shows an increasing tendency so such a *elite* could arise in the future.

4.7.4 Tracking remaining Debian Maintainers

At the time of the release of Debian 2.0 in July 1998 there were 216 voluntary developers contributing to Debian. We have studied how the involvement of these contributors has changed over time. Table 4.40 gives an overview of the number for each release of contributors left from the original group, as well as the number of packages maintained by them. As we can observe from the table, the number decreases steadily, with only 121 of the original 216 contributors (55.8%) contributing to Debian in December 2004. Hence, after six and a half years the half-life value has not been achieved. If the current trend persists, the value of the half-life measure will be around 7.5 years (or 90 months). Taking other releases as the starting point gives similar percentages of abandonment, so this seems to be the common trend. It would be interesting to perform further analysis about which factors influence how long volunteers remain active. There is already evidence that some volunteers face feelings of burn-out [Hertel *et al.*, 2003], but additional studies of human-resource management and motivation in libre software projects would clarify this issue.

The number of packages for which these developers are responsible is also interesting. The initial number of packages maintained by the 216 contributors of Debian 2.0 was 1,101. The number of packages for the developers who remained in Debian 2.1 (around nine months later) rose to 1,351 and then to 1,457 for Debian 2.2 where the maximum number of packages was achieved. For Debian 3.0 it decreased to 1,305 and in the last Debian version, it had similar figures than in their first release (although now with half of the maintainers).

All this means, that there has been a continuous increase in the mean number of packages that remaining maintainers are responsible for. If the number of packages per maintainer was slightly above 5 in July 1998, this number has grown up to nine packages per maintainer in December 2004. It seems also that the mean value keeps on growing but at a lower pace and that it has a tendency towards a value around nine as we can see from the last two releases (which are significantly spaced to assume that a continuous mode has been achieved). Again, we can observe a considerable gain in efficiency as we already found for the number of packages per maintainer (see table 4.39). In this case the gain is of almost 100% in number of packages maintained per maintainer, so the increase for experienced maintainers is above the one found for all maintainers.

Date	Release	Maint	Packages	Pkg/Maint	Median	Mode	Std. Dev	Max
Jul 98	2.0	216	1,101	5.1	3	1 (52)	5.8	50
Mar 99	2.1	207	1,351	6.5	4	1 (38)	7.3	55
Aug 00	2.2	188	1,457	7.8	5	1 (33)	9.2	69
Jul 02	3.0	147	1,305	8.9	5	2 (20)	10.6	65
Dec 04	(3.1)	121	1,091	9.0	4	1 (20)	12.1	83

Table 4.40: *Packages maintained by the Debian 2.0 maintainers.*

Regarding the involvement of maintainers, we can ascertain from the median that there is a general shift towards maintaining more packages as the median value starts with 3 packages for Debian 2.0 and raises up to 5 for Debian 3.0. The mode, on the other hand, shows that the number of maintainers who only maintain a single package decreases quicker than the number of total maintainers (the total number of maintainers drops from 216 to 121, a 56%, while the number of maintainers who maintain only one package does it from 52 to 20, a 38%). The cause for this may be twofold: on one hand those maintainers could have left the project and on the other they could have gotten more involved in it by maintaining more packages. If this is true, maintaining one package can be seen as a *hot* zone where nobody stays long time and where a decision has to be taken: to get more involved in the project or to leave.

The standard deviation gives an idea of the distribution of work. Among this first group we can observe a tendency to have a less equally distributed load of work as we have done before for the whole project. The maximum number of packages that a single maintainer is in charge of grows consequently, from 50 packages in Debian 2.0 to 83 in Debian 3.1. It should be noted that the maximum number of packages of the first three Debian versions under study correspond to a different person than the last two ones.

4.7.5 Researching maintainer experience

In the previous subsection we have tracked maintainers from the Debian 2.0 version to study how their contributions have evolved. In this one we are going to do the opposite; we will take the last Debian release (the Debian 3.1 candidate) and will investigate when maintainers collaborated first in the project. We can see this as a measure of experience in the project as maintainers that entered in the same release will be grouped and analyzed together.

The pie in figure 4.41 displays when currently active maintainers got involved in the project. We have investigated the first contribution for every maintainer of a package in the latest release. In addition to the 121 developers who have made steady contributions since July 1998 (release 2.0), 55 participants got involved before Debian 2.1, and 114 arrived with Debian 2.2. In the last two stable releases, 393 and 554 new maintainers have been identified.

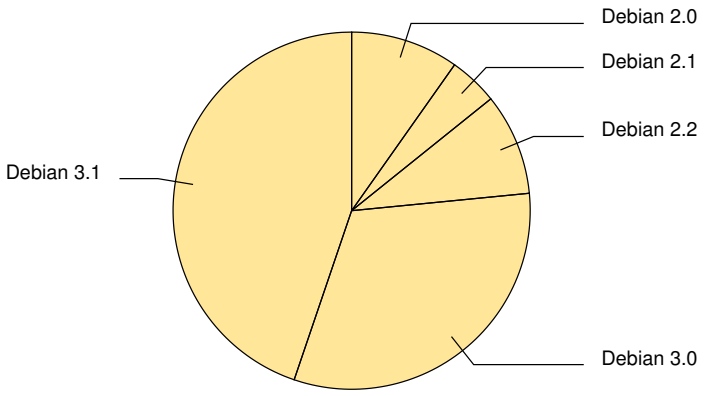


Figure 4.63: *First stable release Debian 3.1 maintainers have contributed to.*

The evolution of the number of packages per maintainer given in table 4.41 provides evidence about the impact of experience in the number of packages maintained in the mean. Noteworthy is that for the first three versions considered, the values range from around 9.0 up to 11.5 packages per maintainer, while in the last two the number of packages is lower. The tendency is in general towards older maintainers having more packages to maintain with the exception of those who entered in Debian 2.1. For this version, we can see a statistical distortion in the mode as it has a value of 5 while the values for all other versions is 1 (or 1 and 2 in the case of Debian 2.2). This has as a side effect a very high mean value for packages per maintainer.

Date	Release	Maint	Packages	Pkg/Maint	Median	Mode	Std. Dev	Gini	Max
Jul 98	2.0	121	1,091	9.0	4	1 (20)	12.1	0.574	83
Mar 99	2.1	55	631	11.5	6	5 (8)	15.1	0.544	81
Aug 00	2.2	114	1,008	8.8	6	1;2 (16)	9.7	0.515	55
Jul 02	3.0	393	2,835	7.2	5	1 (63)	9.5	0.511	121
Dec 04	(3.1)	554	2,221	4.0	2	1 (246)	7.4	0.570	106

Table 4.41: *First release as maintainer for maintainers in Debian 3.1.*

With regard to the median value, we observe that it is also higher for more experienced maintainers, although in this case it is not that clear as with the mean number of packages. Different behaviours for the last two releases can be ascertained; while Debian 3.1 has a median of two with many (up to 246) maintainers only in charge of one package, those maintainers who entered in Debian 3.0 and who are still active have a median value of 5 and few of them maintain a single package. Again, this supports our previous conclusion that maintaining a single package is only a temporary situation.

The standard deviation of the sample does not give us much information in this case. Maybe it stresses the distorted behavior of Debian 2.1 with such a high value, and interestingly enough it shows that the data is more homogeneous as we come nearer to Debian 3.1. This is an expected effect, as *younger* maintainers should have a similar (smaller) involvement while *older* ones may vary more. Nonetheless, the Gini coefficient does not support this finding as it shows values that are very dispersed and with no clear tendency (the highest value is for Debian 2.0 followed nearly by Debian 3.1). This is also the case for the maximum number of packages maintained by a single person which fluctuates from version to version without any predictable direction. In any case, we can conclude that very active maintainers enter anytime in the project, some of them with a surprisingly high involvement (so, for instance, from July 2002 to December 2004 one new maintainer has been able to become the maintainer of 106 packages!).

4.7.6 Packages of maintainers who left the project

When maintainers leave the Debian project, their packages become unmaintained (the Debian project uses the expression *orphaned*). These packages may be taken up by others (*adopted* in the Debian jargon), or will not be present in the next stable release. Table 4.42 contains the ratios and numbers of orphaned and adopted packages between any pair of the releases under study. In this analysis we only consider removed packages for maintainers who have left the project. But it is likely that some other software packages have also been abandoned by maintainers that have not left the project. These cases are not covered by our study.

Release 1	Release 2	Orphaned	Adopted	Adopt/Orph	Orph/Total1	Orph/Total2
2.0	2.1	15	14	93.3%	1.3%	1.0%
2.0	2.2	61	40	65.6%	5.5%	1.5%
2.0	3.0	231	171	74.0%	21.0%	4.5%
2.0	(3.1)	372	251	67.5%	33.8%	3.2%
2.1	2.2	47	31	66.0%	3.0%	1.8%
2.1	3.0	302	220	72.8%	19.4%	5.9%
2.1	(3.1)	493	327	66.3%	31.6%	6.3%
2.2	3.0	281	207	73.7%	10.8%	5.5%
2.2	(3.1)	617	403	65.3%	23.7%	7.9%
3.0	(3.1)	596	383	64.3%	11.6%	7.6%

Table 4.42: *Orphaning and adoption of packages. Each row shows packages present in the older release (first column) and not in the newer ('Orphaned' column), and which of those were adopted. Last columns show the percentages of package 'saved' (adopted to orphaned, Adopt/Orph), and orphaned in the newer release to total in the older (Orph/Total1) and newer (Orph/Total2) releases.*

The table should be read as follows: the first column gives the number of packages in Debian 2.0, 15, for which their maintainers have left the project (column "Orphaned") and that have been adopted, 14, by other (possibly new) maintainers (column "Adopted") in Debian 2.1. This means that the percentage of orphaned packages that have been adopted is 93.3% (column "Adopt/Orph"), so in this case few packages got *lost*. The last two columns help situating the amount of orphaned packages affected, giving the share of orphaned packages in comparison to the total number of packages for each release.

Looking at the rest of the rows in the table, we can see that the percentage of adopted packages is very high: more than 60% for all considered releases. This happens even for releases with a very high portion of orphaned packages (for instance, between 2.0 and 3.1). In other words, even though maintainers who have left Debian between July 1996 and December 2004 were responsible for 33.5% of the packages in 2.0, 67.5% of these packages can still be found in 3.0. We can thus affirm that Debian counts on a natural *regeneration* process and that there is a high probability that the packages of a maintainer who leaves the project are being adopted by other, possibly new maintainers.

Another noteworthy fact is that the adopted to orphaned ratio is always decreasing for a given release. This means that the number of orphaned packages grows more quickly than that of adopted, i.e. there are some packages missing in every new release. Therefore, if a package is unmaintained and falls off the next release it will probably not enter a future release.

4.7.7 Experience and importance

We use data from the Debian Popularity Contest (presented in detail in chapter 3 in subsection 3.6.2) to find out whether more 'important' packages are maintained by more experienced volunteers. Table 4.43 displays the data corresponding to installations and use of packages by developers which are still in the project, and which were already present in the studied releases. In it we observe, for instance, how Debian 2.0 and Debian 3.1 have 121 common maintainers, which are responsible for 1,091 packages. These packages have been installed 919,856 times and 362,249 are regularly used.

Release	CMaint	CPkg	Installations	Votes	Inst/Maint	Votes/Maint
2.0	121	1,091	919,856	362,249	7602.1	2993.8
2.1	176	1,722	1,306,067	498,061	7420.8	2829.9
2.2	290	2,730	2,135,137	805,642	7362.5	2778.0
3.0	683	5,565	3,712,435	1,280,173	5435.9	1874.3
(3.1)	1237	7,786	4,566,601	1,487,246	3691.7	1202.3

Table 4.43: *Installations and regular use of packages. The CMaint column shows how many maintainers 3.1 had in common with the release in the first column, while the CPkg shows the number of packages maintained by them. Columns Installations and Votes give the sum of the packages installed and voted (used regularly) for those packages maintained by common maintainers. The last two columns show the ratios of both to common maintainers.*

If we take the number of installations per maintainer and the number of regularly used packages per maintainer ('Votes/Maint') we are able to answer the question. According to our hypothesis these ratios decrease, which means that more experienced volunteers maintain packages which are installed and used more often. This can be observed through all Debian releases, so this is an evidence that many of the essential components of the Debian system were introduced in the first releases, and that new packages are mostly add-ons and software that are not installed and used that often.

4.7.8 Summing up for Debian

In the last subsections we have studied the evolution of volunteers in the Debian project. We have observed an increase in number of Debian maintainers (from around 200 to over 1200). Another results is that work is more equally distributed than in other libre software projects (with Gini coefficients around 0.5, while usual values range between 0.7 and 0.9), although the growing trend towards inequality.

Then, we have taken those maintainers who were already part of the Debian project in the Debian 2.0 release and have tracked this group. We have noticed how while their number shrank almost to half of its size in six and a half years, the contribution of the remaining maintainers rose significantly doubling the mean number of packages per maintainer. With high probability, this growth in productivity is the effect of further experience and commitment to the project, but also due to the considerable help of new packaging tools that the Debian project has implemented and enhanced during these years.

Throwing a look at the procedence of maintainers who maintain in package in Debian 3.1 we have seen how *older* maintainers maintain in the mean a larger number of packages than *younger* ones. Noteworthy is a fact we have observed in several of our analysis: maintaining a single package is usually a temporary situation. Those maintainers either leave the project or get involved by maintaining more packages.

We have also studied what happens to those packages that are left unmaintained by Debian maintainers who leave the project. We have seen that there exists a high rate of take-over. Finally, we have used data from the Popularity Contest to see that the weight of packages maintained by more experienced maintainers is higher that that of those who have entered the project recently. We have concluded that this indicative for *older* maintainers taking over essential packages that were part of the Debian distribution from the beginnings.

4.7.9 Evolution of the core group

The study of the Debian project has made the analysis of a team formed entirely by volunteers possible. We have investigated from the beginnings towards the end (focusing on those who contributed to Debian 2.0) and from the end towards the beginnings (taking into account those who maintain packages in Debian 3.1). But there is a gap left in between which we have not studied in detail and which we will do next in our study on the evolution of the *core group*. The goal is to see if we can find several leading generations in libre software projects.

As previously noted in the description of the methodology, we base our analysis of the evolution of the *core group* of libre software projects in dividing the project life-time into ten equally large intervals. This means that we will have different interval lengths depending on the project. As the projects we have selected as a case study have at least three years of development, the minimum length of the time interval is more than three months.

For each interval we consider the activity measured in terms of commits performed on the CVS repository. The most active 20% of all commiters (rounded by excess) for that interval is what we consider the *core group*. So, we should have ten different *core groups* identified, one per time interval. It should be noticed that the composition of the core group depends on the number of commiters on a given interval; if only 10 commiters participate in the first interval considered, our core group will be composed of two persons. If in the last interval the number of participants is 18 commiters, the core group will have 4 members.

The analysis technique we will use is based on visualizing the contribution of the *core groups* over time. This means, that we will identify the *core group* in the first time interval and then plot its contribution not only in the first time interval, but also in all the other intervals. Then, we will follow on with the *core group* in the second interval, plotting the aggregated contribution of its members from the first to the last interval. This process will be repeated for every *core group*, yielding in ten curves (one per *core group*) that go from the first to the last interval.

Some important considerations in order to fully understand the visual information that the plots provide should be taken into account. One is that *core groups* may have members in common. This may happen if a commiter is part of the most active 20% in several time intervals. Visually, this is sometimes not easy to identify at first as what is plotted is the aggregated contribution of all the members of a *core group*; but for our intention of recognizing generations such a detailed analysis is not required as we will see next. In some cases, subsequent *core groups* will be composed by the same persons, which is easy to identify visually as curves have then exactly the same shape.

We have plotted the resulting data in three different graphs, which differ in how we plot the contribution of the various *core groups* (contributions are always displayed in the vertical axis, while

the horizontal axis corresponds to the ten intervals):

- Absolute graph: it displays the absolute number of commits by each *core group* (vertical axis) for each interval over time (horizontal axis).
- Aggregated graph: it displays the aggregated number of commits by each *core group* (vertical axis) since the beginning of the project vs. time (horizontal axis). This graph is just the integral of the absolute graph.
- Fractional graph: it displays the fraction of the total commits during an interval done by each *core group* (vertical axis) for each interval vs. time (horizontal axis). This graph provides the same information as the absolute graph but normalized by the number of commits that has been done during each period.

Our experience is that, in general, the fractional plot provides the easiest way for the analysis of the evolution of the *core group*. Nonetheless, it should be contrasted with the information that the absolute and the aggregated graph display, as for instance, periods of less or much activity are impossible to be identified by the fractional graph.

When looking at the resulting graphs, it is usually simple to inspect whether the same core group rides the project from its beginning to the current days. The following case studies will help us considering the convenience of this methodology.

The number of libre software projects to which we have applied the methodology is 21. We have selected among them three examples which could be assumed as canonical examples for different types of behaviours: a case where there are no generations (The GIMP), another one where several generations appear (Mozilla) and finally one that yields results which cannot be assigned to neither of the previous groups (Evolution). The rest of case examples, up to 18, will give us an idea of which of the three behaviours is more frequent.

4.7.10 Observations on The GIMP

The GIMP is probably the canonical example of a project with *code gods* or a single generation leading the project. Table 4.44 displays a small summary of the most important facts related to our analysis. The size of The GIMP is above half a million lines of code, but there is evidence that it has stagnated since several years ago and does not follow the typical linear growth which we have found to be most frequent in libre software applications (see section section 4.2). Regarding activity, with over 100,000 commits, The GIMP can be considered a very active project.

Project	The GIMP
Size	557 K
Commits	125,590
Start	(Dec 97)
Ver 1.0	Jun 98
Interval	7.5 months
Generations	Code god

Table 4.44: *Summary of the most important facts for The GIMP project.*

Although The GIMP is prior to December 1997 (that is why the starting date appears in brackets in table 4.44), it was only then when it was introduced into the GNOME CVS repository. So, we have only data from that moment onwards. The version 1.0 of The GIMP was released in June 1998; we can consider The GIMP a stable and mature project. The length of the intervals in which we have divided the project is slightly above half a year (7.5 months).

Figure 4.64 displays on the left the absolute graph of commits for each *core group* and for each interval. We can see that there are at least two groups (two generations), as it seems that the *core group* in the beginnings is different from the one found in the rest of the intervals. In any case, the

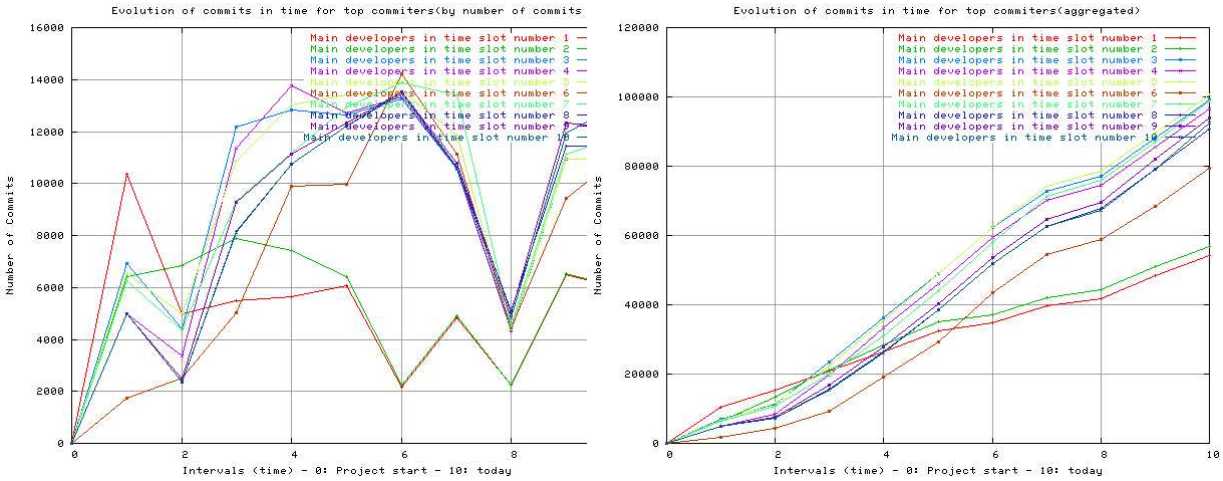


Figure 4.64: *Right: Absolute graph for The GIMP project. Left: Aggregated graph for The GIMP project.*

members of these *core groups* do not all leave the project as their contribution in subsequent intervals is in the thousands.

A detailed study of the developers who compose the core groups yields that one of the most active developers is present in all core groups. The second and third most active developers enter in the third interval (which starts around mid-99) and keep in the project until today.

The plot on the right in figure 4.64 strengthens this perception. Here the commits each *core group* has performed are displayed in an aggregated manner. Parallel curves are indicative for *core groups* for which the most contributing developers are the same. We can easily identify therefore the first two *core groups*, when the most active developer was the leading person in the project, as their curve is below the rest of the curves for later intervals. On the other hand, the shape of the curves from the *core group* from the third interval onwards indicates that they only differ in the number of members of the *core group*, which as we have seen is variable as it depends on the total number of contributors in any given interval.

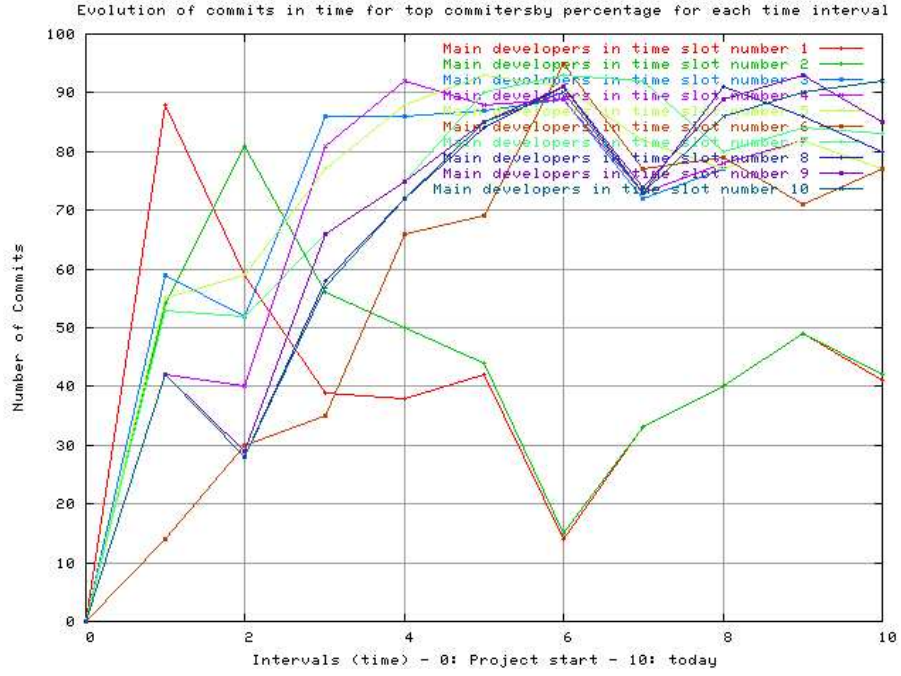


Figure 4.65: *Fractional graph for The GIMP project.*

The fractional graph, depicted in figure 4.65, gives us further information for these type of analysis. Now the vertical axis has been normalized to 100% of the total commits during a given interval. By definition the maximum in each interval will always correspond to the *core group* identified in it. In the case of a project with *code gods*, the other *core groups* should be near that maximum (or at the same level) as the composition of the *core groups* does not change much. In the case of The GIMP this is true, but not for the first two intervals, as we have seen in the previous plots. We see, that there is a fall of the contribution of the two first core groups, especially in the sixth interval where it lies under 20%.

Interestingly enough all core groups have a share that is bigger than 80% in their intervals, with values above 90% for those core groups after the third interval. This gives us again evidence about the inequality that exist in the contributions of libre software projects. We knew by now that something near to a 20%-80% Pareto distribution happened in libre software projects; this graphs demonstrates that, at least for The GIMP (we will see below if this happens in other other projects), this is general even for (sufficient large) time intervals in the project.

4.7.11 Observations on Mozilla

We have selected the Mozilla Internet suite as an example of a libre software project where multiple generations can be identified. Mozilla is a well-known libre software project, the follow-up of the Netscape Internet suite. Table 4.45 sums up the information about Mozilla that is interesting for our analysis. Mozilla is a multi-million project, with over three million source lines of code. The CVS activity around the project is above 650,000 commits, over five times larger than The GIMP.

Project	Mozilla
Size	3,414 K
Commits	663,454
Start	(Oct 1998)
Ver 1.0	Jun 2002
Interval	6.5 months
Generations	Multiple

Table 4.45: *Summary of the interesting information on Mozilla.*

Mozilla was released in 1998 by Netscape in order to attract the attention of libre software users and developers. Although its beginnings were not very promising, it seems that the project has surpassed those initial concerns and finally, in June 2002, version 1.0 was released. Following our methodology, we have ten intervals of 6.5 months each, slightly below the 7.5 months that we have used for The GIMP.

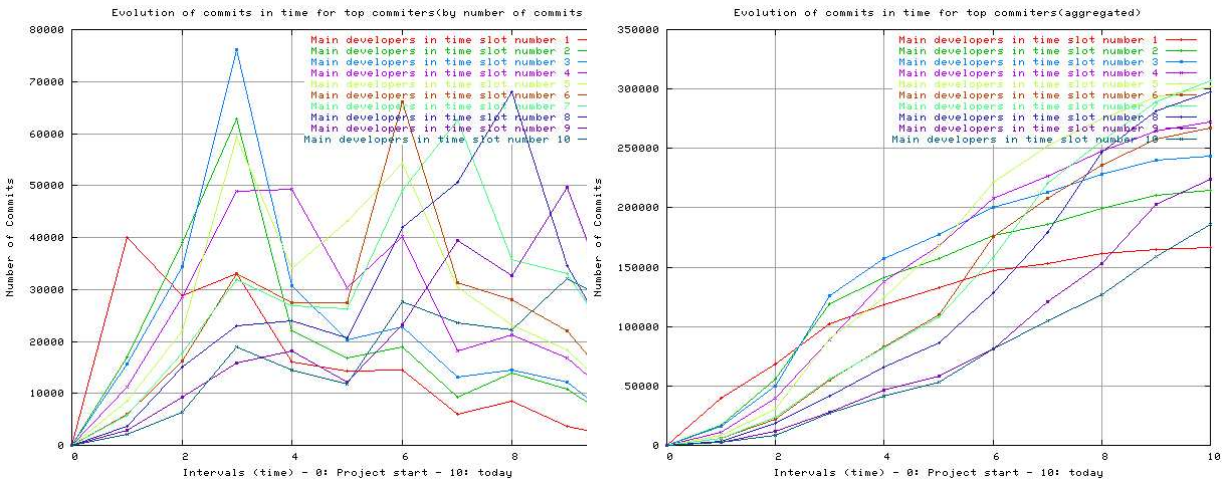


Figure 4.66: *Right: Absolute graph for the Mozilla project. Left: Aggregated graph for the Mozilla project.*

Figure 4.66 groups the absolute (left) and aggregated (right) plots of the ten *core groups* for each interval. At first sight, we can already observe that there exist many differences between these plots and the corresponding ones of The GIMP project. The absolute graph gives us interesting information about the over-all activity in the repository. We can see that in the first two intervals and for the fifth the peak of the *core groups* is not that high, a fact that is indicative for fewer activity. Attending to the aggregated graph, the number of curves which follow their own way (i.e. are not parallel one to each other) is larger. In other words, the composition of *core groups* varies more frequently than in The GIMP.

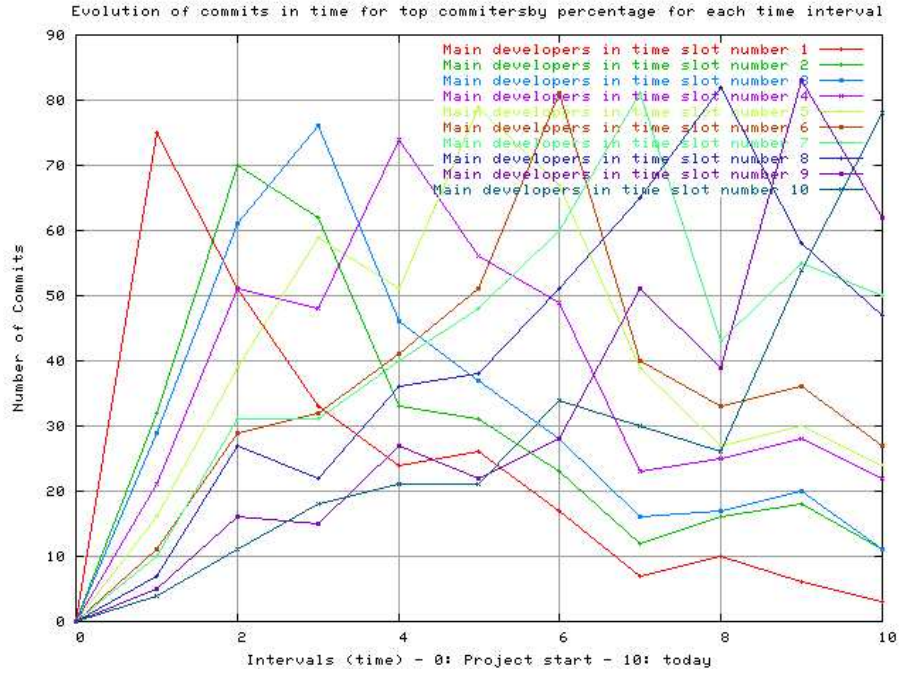


Figure 4.67: *Fractional graph for the Mozilla project.*

Again, the figure that will provide more information is the fractional one (see 4.67). We can clearly identify several generations in it. All of these *core groups* achieve peak values of over 75% in their intervals (over 80% in later ones).

Noteworthy is how the *core group* in the last interval contributed already in the early stages a small amount of commits (around 5%). Its contribution raises then almost continuously (the sequence in the ten intervals is the following: 5%, 11%, 18%, 21%, 21%, 34%, 27%, 55% and finally 78% where it is the leading core group). The core group that achieves its peak contribution in the first interval has an opposite trend, declining as time passes. In between we find several core groups that have both behaviours; they achieve a peak and have a declining part afterwards. If we compare this figure with the corresponding one for The GIMP, we conclude that the more chaotic a fractional graph is, the more generations exist. *Code god* projects have a tendency to curves in parallel, while projects with several generations display curves crossing each other.

4.7.12 Observations on Novell Evolution

Finally, we have selected a project which exhibits a mixed behavior between *code gods* and multiple generations. This is the case for Ximian Evolution (currently renamed to Novell Evolution), a groupware solution for the GNOME project. Table 4.46 displays the most important information about Evolution: it is a medium-sized application with around 200 KSLOC. From our software evolution study in section 4.2 we know that it had a smooth growth and some code restructuring in the past. The amount of commits is in the order of magnitude of The GIMP.

The history of Evolution will give us further insight of the results. Evolution started as a community-driven project in December 1998. By the end of the year 1999 it was chosen by a small

start-up company called Ximian as a key application in its strategy. This meant that professional developers started to work on the project, changing its governance structure to suit the one found for a company-driven project. Version 1.0 was delivered ending 2001. The time for each interval is around 6.4 months, similar to the interval lengths for Mozilla and The GIMP.

Project	Evolution
Size	208 K
Commits	92,333
Start	Dec 1998
Ver 1.0	Dec 2001
Interval	6.4 months
Generations	Composition

Table 4.46: *Summary of the interesting information on Evolution.*

The absolute graph on the right of figure 4.68 gives a clear idea of the lower activity that prevailed in Evolution before Ximian developers took it over in the third interval. Then an increase in activity is reported for several years (achieving up to 16,000 commits each interval), declining activity in the last intervals to values near 9,000 commits per interval. The aggregated graph on the right supports our findings: the first two core groups (which are identical in their composition) do not contribute after the initial periods, while the third one shows to be a combination of the first two with some new developers that have prevailed from then. The other *core groups* show the typical *code god* behaviour with almost parallel curves.

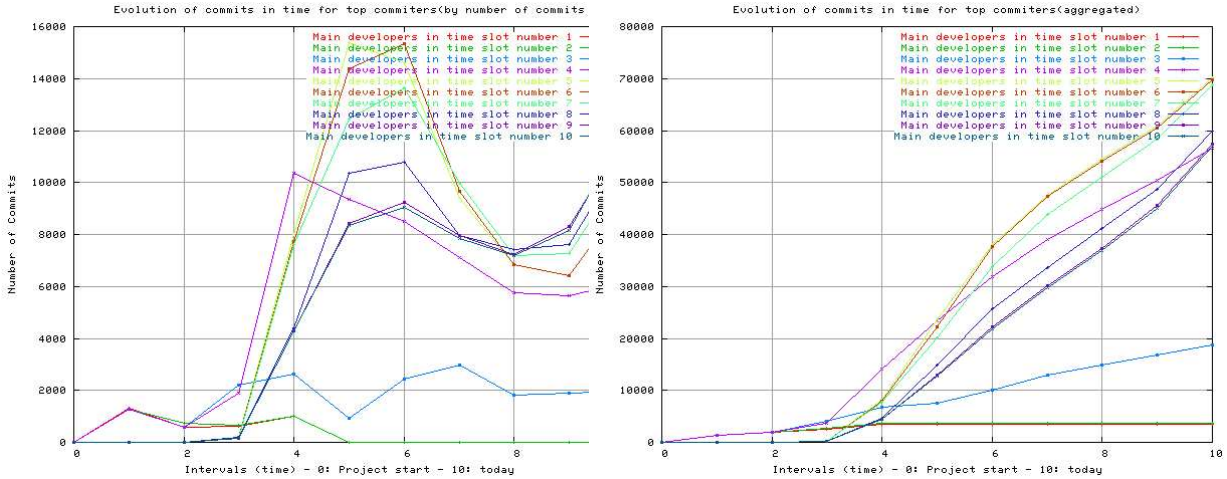


Figure 4.68: *Right: Absolute graph for the Evolution project. Left: Aggregated graph for the Evolution project.*

The fractional graph in figure 4.69 is the one that displays again the mixed behavior best. In the first three (maybe four) intervals, a multiple-generations behaviour may be observed. From then on, the *code god* pattern is clearly identifiable with a small reminiscence from the past in the curve that achieves its peak in the third interval and that does not disappear in the following intervals.

4.7.13 Observations on other libre software applications

This subsection is devoted to state which one of the three described behaviours (*code god*, multiple generations or mixed) is the most common pattern in large libre software applications. Therefore we have selected a similar set of applications than those for our software evolution analysis in section 4.2. The case studies are part of GNOME (Gnumeric, GTK+, Galeon and Nautilus), KDE (kdelibs, KOffice, kdepim, kdebase, kdenetwork and KDEvelop), Apache (jakarta-commons, xml-xalan and ant), Mono (mono and mcs) and FreeBSD. In the case of FreeBSD, we study the *src* CVS module which contains the kernel and many of the applications that it ships.

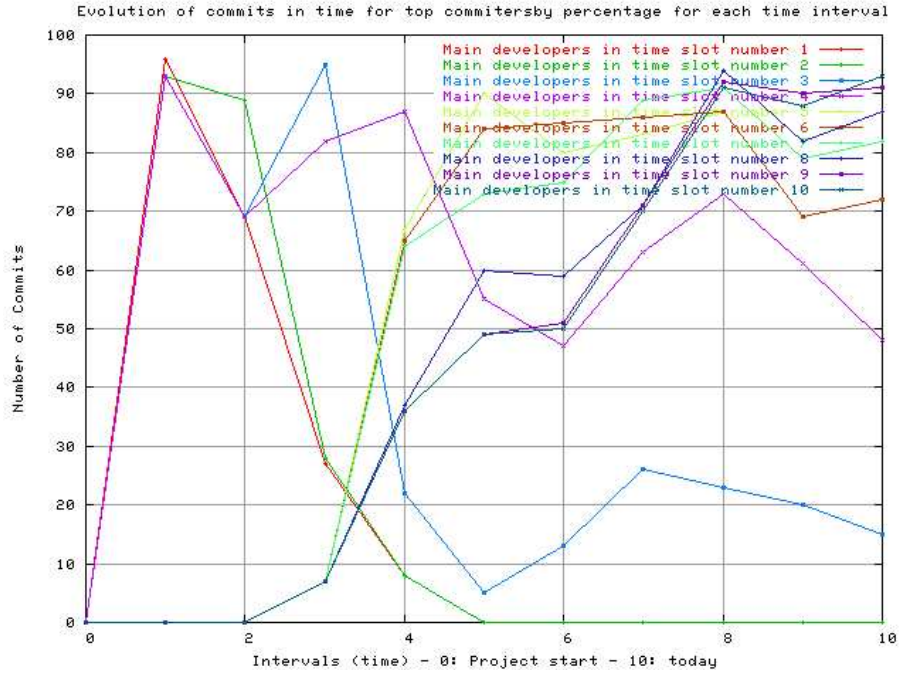


Figure 4.69: *Fractional graph for the Evolution project*

Table 4.47 gives a summary of the selected projects. The starting year of the applications under study range from December 1993, where the first commits to the FreeBSD projects were performed, to June 2001, for the Mono project. With the exception of jakarta-commons, all have achieved the 1.0 version, so we can assume that they are stable. The length of the intervals depend, of course, on the starting date of the repository. Hence, we have intervals that go from one year to those that almost correspond to three months (for instance, for mono and mcs, the two younger applications). A column with the project size, in number of commits and commiters, has been added to give additional insight of the applications and to demonstrate that they can be considered as *bazaar*-driven applications.

As fractional graphs have proven to offer the best visualization, they are the only ones that will be displayed. We have located them in two 2x4 matrices, specifically in tables 4.48 and 4.49. Projects have been ordered as in table 4.47, with those projects with multiple generations first, then those that exhibit a mixed behaviour and finally the projects that could be classified as *code god* projects.

After a visual inspection, we have decided to group eight of the projects as exhibiting multiple generations, six that could be classified as mixed and finally only two projects which behave as having *code gods*. Further research should study how much the selection of the interval length affects the (visual) results. Our experience so far proves that selecting time slots larger than five to six months is sufficient to identify the existence of several generations, but it does not allow to recognize the total number of them.

On the other hand, our sample is composed of large libre software projects. An interesting future research activity could be to investigate the results obtained from applying this methodology to projects with a smaller number of contributors.

Project	Start	Ver 1.0	Size	Interval	Commits	Commiters	Type
FreeBSD (src)	Mar 93	Dec 93	1500K	12.1	554,764	352	MG
kdelibs	May 97	Jul 98	615K	8.3	217,961	441	MG
jakarta-commons	Mar 01	-	429K	3.3	39,370	72	MG
mcs	Jun 01	Jun 04	1081K	2.7	32,566	114	MG
kdenetwork	Jun 97	Jul 98	293K	8.1	98,282	332	MG
kdevelop	Dec 98	Dec 99	386K	6.2	69,890	152	MG
koffice	Apr 98	Jan 01	780K	7.1	172,564	247	MG
kdepim	Jun 97	Jul 98	512K	8.1	93,632	284	MG
gtk+	Dec 97	Apr 98	388K	7.7	68,279	265	MB
galeon	Jun 00	Dec 01	90K	4.5	31,153	110	MB
xml-xalan	Nov 99	Oct 00	337K	4.9	54,267	32	MB
kdebase	Apr 97	Feb 99	362K	8.3	330,009	450	MB
ant	Feb 00	(Aug 03)	120K	4.7	43,955	33	MB
nautilus	Feb 98	May 01	200K	7.3	63,760	236	MB
gnumeric	Jul 98	Jun 02	229K	6.9	81,019	166	CG
mono	Jun 01	Jun 04	222K	2.7	11,936	91	CG

Table 4.47: *Summary of the findings for a generations analysis applied to the projects listed in the first column. Start is the starting date of the CVS, Ver 1.0 the date of version 1.0 if available, size gives the size of the software in SLOC, interval gives a tenth of the life-time (in months), commits the total number of commits, commiter the total number of commiters and generations their type (MG = multiple, M = mixed behaviour, CG = code gods).*

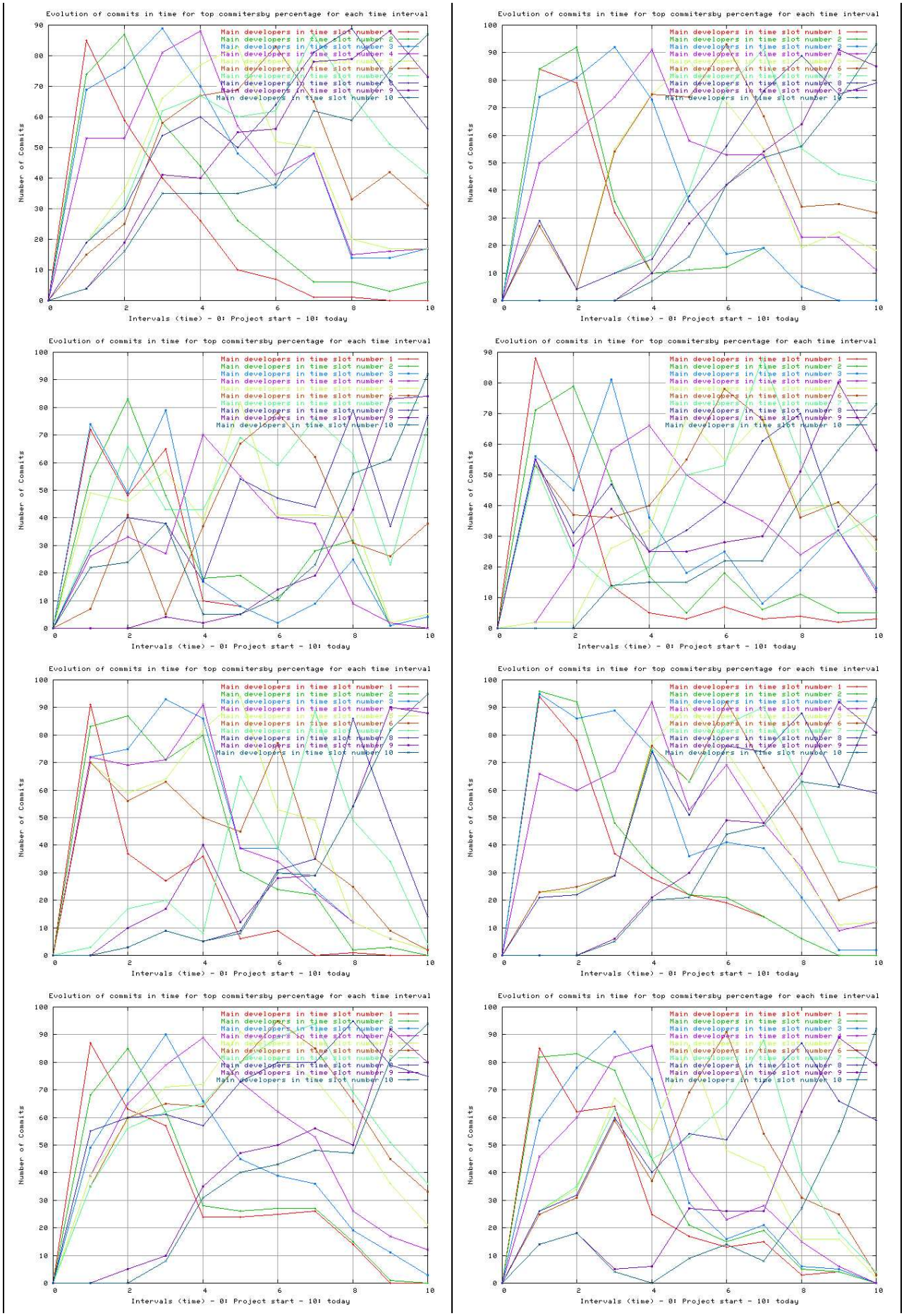


Table 4.48: *2x4 matrix with fractional generation plots for 8 libre software systems. Projects with heavy generational turn-over have been situated at the top. More information can be found in table 4.47.*

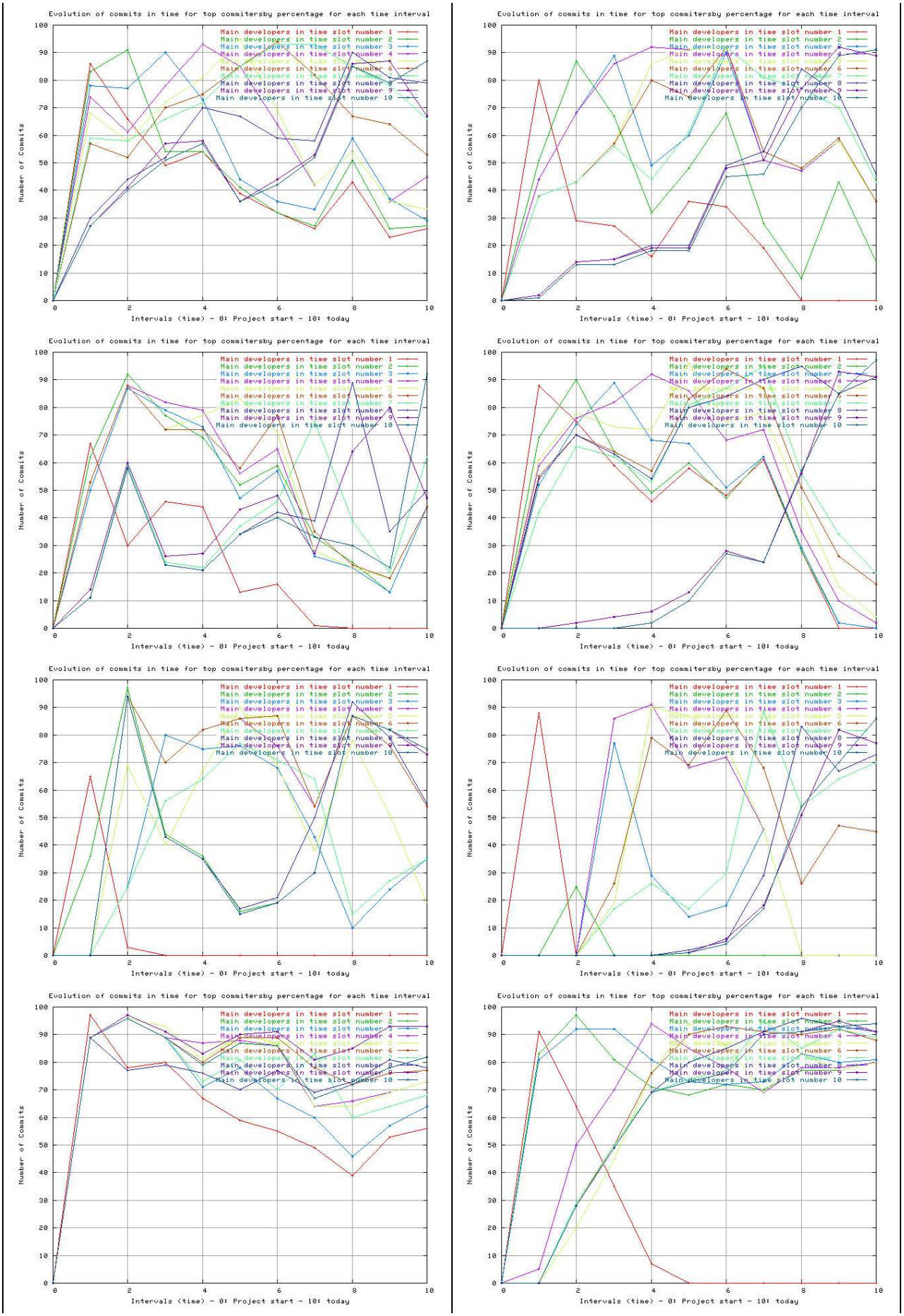


Table 4.49: *2x4 matrix with fractional generation plots for 8 libre software systems. Projects with heavy generational turn-over have been situated at the top. More information can be found in table 4.47.*

4.8 Membership integration

In the previous section, we have demonstrated that in some projects developers enter the project in later stages and achieve to belong to the *core group*. But the analysis has been performed in a aggregated way. As we have longitudinal data for developers, we are in a position where we could identify these developers and study their evolution individually. That is precisely the goal of this section. We have selected a large libre software project, GNOME, and have identified those developers who entered the project once it was mature. Then, we filter all those who have contributed with an important contribution to the project (assuming hence that they form part of the *core group* of any of the applications in GNOME) and analyze the process.

The conceptual model on which we will base our methodology is the onion model [Crowston *et al.*, 2003a]. It says that projects have a small number of persons, the *core group*, who are very committed to the project, and that perform the vast majority of tasks [Mockus *et al.*, 2002]. The *core group* is surrounded by an ample community that can be layered by their contributions in several subgroups from casual developers to users. Our supposition is that members who enter the group start as users, get then involved as casual contributors and, after some time, may end belonging to the *core group*.

The onion model is a static representation of the structure of a libre software project. With this analysis we are introducing dynamicity into it, assuming that this structure is not stable (at least not for large periods of time) and, therefore, changes as newcomers enter the project and collaborate on it. Our assumption, graphically depicted in figure 4.70, is based on developers following a trajectory that goes sequentially from the outer positions (within the user base) to the inner positions (forming part of the core group).

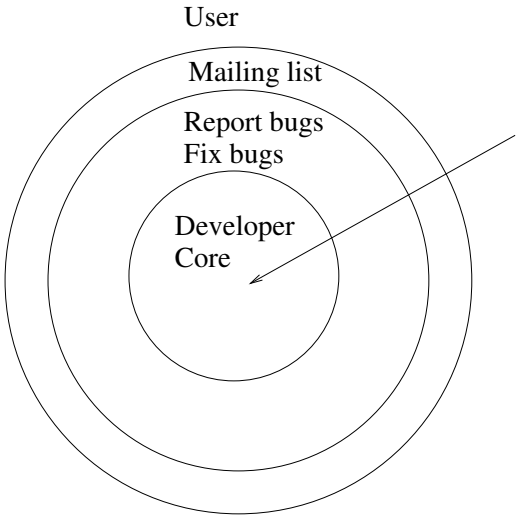


Figure 4.70: *The Onion model.*

4.8.1 Goals

Our main goal is to verify if the described sequence can be observed in real cases, and under what circumstances this occurs. Therefore, we will characterize the sequence by observing the activity of developers in mailing lists, bug tracking systems and versioning system repositories.

In addition, the empirical study is designed to quantify other aspects of this process, measuring the time that it takes to developers to move from one step to the next. We will later use these measures to identify common patterns, and correlate them with specific characteristics of the developers such as if they are volunteers or developers hired by companies.

More specifically, this investigation tries to answer several questions. The first one is concerned with the onion model and if it can be used as a good representation of the joining process of a libre software developer. Our study is based on the assumption that developers start in the outskirts as advanced users and end as part of the *core group* after an integration process that includes participating

first in the mailing lists, then in the submission and correction of bugs and finally adding code to the project through the versioning system. Our intention is to empirically validate this idea and to find out how many of those who have achieved being part of a *core group* really follow this sequence.

If the onion model is a good representation, measuring how long it takes for developers to go through the different steps would be the next step in our analysis. Answering this question has practical implications for libre software projects as it allows to know how much time it takes to achieve a position in the *core group*. Although beyond of the scope of this thesis, this may have important lectures from other points of view; in this sense, we assume that the integration process proceeds in parallel to a software comprehension process as contributions in later steps of the integration process require more knowledge of the project and of the environment. Finding ways of lowering the *cost of entry* that these comprehension supposes is surely one of the key issues any (libre) software project is interested in.

But we expect that the set of programmers that has followed the integration process is a heterogeneous group and that we can infer interesting information from the differences that arise from their integration processes. So, do different patterns (different sequence or timing) exist? We will especially focus on those who have different sequences than the ones assumed and on those whose timing in shows a different behaviour.

Furthermore, if different patterns are identified, how does this depend on the *nature* of the developer? Is the integration process the same for volunteers and employees? In this sense, a very interesting issue is the *nature* of the developers that we identify as having reached the *core group* status. It is known that many companies cooperate with the GNOME project, some of them by hiring developers that are very active, others by integrating their own teams in GNOME. We assume that the integration process for such kind of (professional) developers differs from the one for volunteer developers.

Finally, it should be noted that one of the topics of this thesis has been to present the various publicly available data sources found in libre software projects. We have discussed in section 3.7 how to integrate data obtained from several sources and, although in a very limited way, this analysis uses the described techniques and presents some results that should strengthen the interest for further research on this issue.

4.8.2 Methodology

To reach these goals, our study focuses on a certain population of contributors of a well known libre software project (GNOME), who joined the project well after it started in 1997. Using publicly available information (in mailing lists, bug reports and CVS logs) we track their activities during a long period of time, and analyze them. From this analysis, we characterize different patterns of entering the project and identify correlations between each of them and the volunteer/professional status of developers.

Once we had chosen GNOME, we had to specify a set of criteria in order to obtain a sample of developers suitable for our study. We require developers which are mainly source code developers, as our process is tight to *pure* software development. This means, that we are not considering other development-related tasks as translations or documentation. We want developers whose main work is programming in a programming language and who have to comprehend both the current software structure and the project governance and organizational issues. The developers of our sample should have gone successfully through the joining process once the project has passed its initial stages; this avoids having those who founded the project which of course did not need to integrate themselves into it. On the other hand, this also requires the developers in our sample to face existing source code and project structures which they have to understand. Besides entering *late* into GNOME, the developers we are looking fore should still be active to ensure a reasonable period of activity, ensuring that way at least three years of data that we may obtain from them.

Hence, our methodology is based on filtering the GNOME user base for a certain type of developers. We have done this following these premises:

1. A developer starts as a passive user. At this stage the developer occasionally visits the project's

web site, and maybe reads mailing lists. Since at this moment the developer will not submit any e-mail message nor bug reports, activity cannot be tracked.

2. If the developer becomes familiar with the project (advanced user), he may occasionally send some messages to mailing lists (this is activity that can be tracked). At this point our integration process has started, since relations with other members arise even if they are as weak as occasional e-mail exchange. The comprehension of the project (at least of the software and a notion of how it is organized) starts usually to be acquired at this stage.

3. Having at least some partial knowledge of the project, the developer may start to report bugs and participate in the bug-tracking system. Maybe he even fixes some of the bugs. The relationships the developer establishes with occasional and *core group* developers through frequent collaboration in the bug-tracking system are tighter than previous ones. The activity in the bug-tracking system is information that we can track.

4. If the contributions are meaningful, usually the next step for developers is to be given an account with write access to the versioning system repository. A developer will then be able to directly commit code. His contributions will appear in the CVS logs, so we can obtain and analyze them.

5. If the developer collaborates actively at a high level during a certain amount of time in the project he may eventually be considered as a part of the *core group*. We have considered 1,000 commits as the value that developers have to commit in order to be considered as part of a *core group*.

Summarizing, for any developer we may track his activity in the mailing lists, in the bug-tracking system and in the CVS versioning repository. Any activity beyond these data sources cannot be measured. We will devote the next subsection to specify these conditions, which of course are based on the criteria that have been presented above.

To obtain the real sample of developers that we want to study (those who have got integrated into the project and have become part of the *core group*), we specify a set of conditions, preferably deterministic and numerical so that we can empirically satisfy them. In detail, we have studied the rules given above and have determined the following set of objective criteria to obtain this list of developers:

1. The developer has to have participation in mailing lists, the bug-tracking system and the CVS repository.

2. The developer should have CVS write permission, i.e. a CVS account in the CVS repository.

3. The developer should have done his first commit at least one year later than when the modules he is working on was set-up. This restriction ensures that the developer had to understand an already large code structure when he enters the project.

4. The first commit to CVS should be not later than April 2001. This allows to have enough data (three years, as the data collection was performed April 2004) from the developer for this study.

5. The last commit to CVS should be in the year 2004, meaning that the the developer has to be an active one at the time of the data collection.

6. The main contribution of the developer to the project has to be source code. So, a developer should have more than 50% of its commits done to source code files in CVS.

7. The number of commits to the CVS should be greater than 1,000, meaning that a developer is (or at least has been) very active in the last years.

The analysis starts by collecting the data from the three data sources and merging authorship traces from the different sources. Applying the aforementioned methodology we got a first sample of developers who meets the selection criteria. However, we have to screen this set carefully to discard

some developers who in fact fail to comply with all the criteria. We then study which of those patterns fit the assumptions of the onion model, and try to identify common characteristics and parameters of the various groups of developers found in the previous step. All this process, and the results found, are described in detail in the next subsections.

4.8.3 Merging authors and screening the sample

After retrieving the data from the data sources considered for this study (following practices that have been described in chapter 3), we obtain the following data set:

- 464,953 messages from 36,399 distinct e-mail addresses have been fetched and analyzed.
- 123,739 bug reports, from 41,835 reporters, and 382,271 comments from 10,257 posters have been retrieved from the bug tracking system.
- Around 2,000,000 commits, made by 1,067 different commiters have been identified from the CVS versioning repository.

To these data, we apply the merging techniques, described in section 3.7, that identify the various forms a developer appears in several sources. As a result, we obtain 108,170 distinct identities with 47,262 matches, of which 40,003 are distinct. Using this information, we are capable of finding 34,648 unique persons. This process has been statistically verified by selecting a sample of identities, looking manually for matches and comparing the results to the corresponding entries in the Matches table (see section 3.7.2 for further details).

In any case, from the over 30,000 identified developers only 1,067 are of interest for us, as that is the number of those who have write access to the CVS versioning repository (one of the conditions mentioned above). We study the developers with write access to the GNOME CVS repository. After automatically confronting their patterns of activity in CVS, mailing lists and bug report system with the selection criteria, we got a set of 32 which could meet those criteria.

At first, it seems that the amount of developers that we have identified (which lies around 3% of the total) is very small. Restriction number 7 (the one that supposes a high amount of contributions for being part of the core group) shrinks the number of commiters to 295 which is really near to the 20% assumption that we have used in the previous study on *core groups*. The number of commiters with over 1,000 commits that started to contribute to the CVS before April 2001 (restriction 4) is 201, so some of those who have been very active entered the project too late to obtain sufficient data about them. 141 of the 201 commiters left the project before 2004, so they do not fulfill the condition of being still active in the project (restriction number 5). Finally, we get only 32 commiters that have worked on modules and that were active at least one year before they started to contribute (restriction 3).

Further analyzing this set, by identifying and screening developers one by one, we find that 12 of them do not really meet the selection criteria for several reasons. For instance, we have found developers that have been active in the CVS earlier than the data suggested, because at that time they were using a different username; others have contributed to a module which had neglectable activity during their first year; and even one who had no participation in Bugzilla, the bug tracking system of the project. This screening, therefore, has lead us to a final sample of 20 developers.

4.8.4 Identifying patterns and common characteristics

We can classify the activity patterns of the 20 developers in three groups. The first one, composed of 7 developers, includes those developers who have clearly followed the predicted sequence according to the *onion model* (this group will be referred from now on as "Group 1"). Developers in this group follow a slow, gradual joining process. As expected, the first activity we track for any of them is an e-mail message. Later, they report a bug, fix a bug, and they finally get access to the CVS repository and commit a change there. This process lasts usually between two and three years.

The second group (referred from now on as "Group 2"), composed of 12 developers, has a pattern that does not fit the predicted sequence. On the contrary, the members of this group display an

almost simultaneous start-up in the mailing lists, the bug tracking system and the CVS repository. Developers in this group follow a fast integration process, with sudden activity at all levels, usually completing the whole process in less than one year. Four of them even begin their participation in all systems almost at the same time.

Finally, a single developer has a completely different joining pattern (referred from now on as "Rest") than the described for the previous groups. This seems to be a sort of *rara avis*: it took him almost three years from its first message to a mailing list to the first commit to CVS. And two years later he began to be active in the Bugzilla system. Table 4.50 gives some statistics about this developer (which we will refer to as developer 20).

	1st message	1st report	1st fix	1st commit	commits	messages	bugs	comments
20	11/13/96	07/18/01	02/24/01	09/02/99	5753	8673	6	19

Table 4.50: *Some statistics about the selected developers (rest)*

We have computed some progress metrics for each group. Considering the date of their first message to a mailing list as the beginning of the process, we calculate the time between the first participations (report and fix) recorded in the bug tracking system and the first commit in the CVS repository. The aggregated results of these metrics for each group are shown in table 4.51. The detailed results for each developer will be discussed later.

Samples	*TT* commit	sd. dev.	*TT* bug rep	Sd.Dev.	*TT* bug fix	Sd.dev.
Whole sample	13.5	19.9 (147.2%)	21.9	18.3 (83.5%)	22.4	16.5 (73.5%)
Group 1	29.6	17.6 (59.5%)	29.4	16.2 (55.1%)	35.0	13.7 (39.1%)
Group 2	0.2	6.8 (3467.1%)	12.6	12.5 (99.6%)	13.1	8.7 (66.5%)

Table 4.51: *Progress metrics for each group (TT, time-to 1st commit/bug report/bug fix, are given in months).*

In any case, with these results we can already answer one of the questions we have raised. For the given sample, Group 1 complies with the predictions of the onion model, while for Group 2 the pattern is clearly not compliant.

In the next subsections we will enter more in detail about the data shown in table 4.51, but before we will deepen in the different groups. Once we have obtained a first discrimination of our sample, our aim is to find a parameter that could correlate with this behaviour. Fortunately, we have easily found one: the status of the developer as a volunteer developer or as an employee hired by a company to work on the project. To obtain this information, we have performed web searches and have contacted persons of the GNOME community. For sure, we could have contacted the developers themselves given the small size of the sample, but for the purpose of our study we wanted to remain as non-intrusive as possible.

Developer Nature	Number
Volunteers	8
Employees	8
University staff	4

Table 4.52: *Nature of the developers in the sample*

We have found that from the sample of 20 developers, 8 are basically volunteering in their spare time, 8 are employees hired by companies to work on GNOME and 4 are working as staff at universities (see table 4.52). Interestingly enough, seven out of the eight volunteers compose Group 1, while all those hired by companies or working at universities integrate Group 2 (more information in table 4.53). The 8th volunteer developer is the one that we could not classify in neither group. Thus, at least for the GNOME project, developers that achieve to become part of the *core group* following a process as predicted by the onion model are volunteers. This conclusion may not be surprising at all if we

consider that usually developers hired for working on a libre software project are supposed to work on it from the day they are contracted.

Name	Volunteers	Employees	University staff	Total
Group 1	7	0	0	7
Group 2	0	8	4	12
Rest	1	0	0	0

Table 4.53: *Groups by nature of the developers*

In the next subsections we will further analyze each of the identified groups.

4.8.5 Group 1: progress according to onion model

We want to analyze the length of the various joining steps for this group. Table 4.51 demonstrates that the standard deviations are relatively small. This is a clear indicator of temporal coherence in the timing pattern of the developers. The table also shows that the time to report a first bug is slightly shorter than the time to the first commit. This means that developers get write access to the CVS right after issuing bug reports. Figure4.71 provides the same information for four selected developers of this group, but in a visual manner while table 4.54 displays data for all the members of this group.

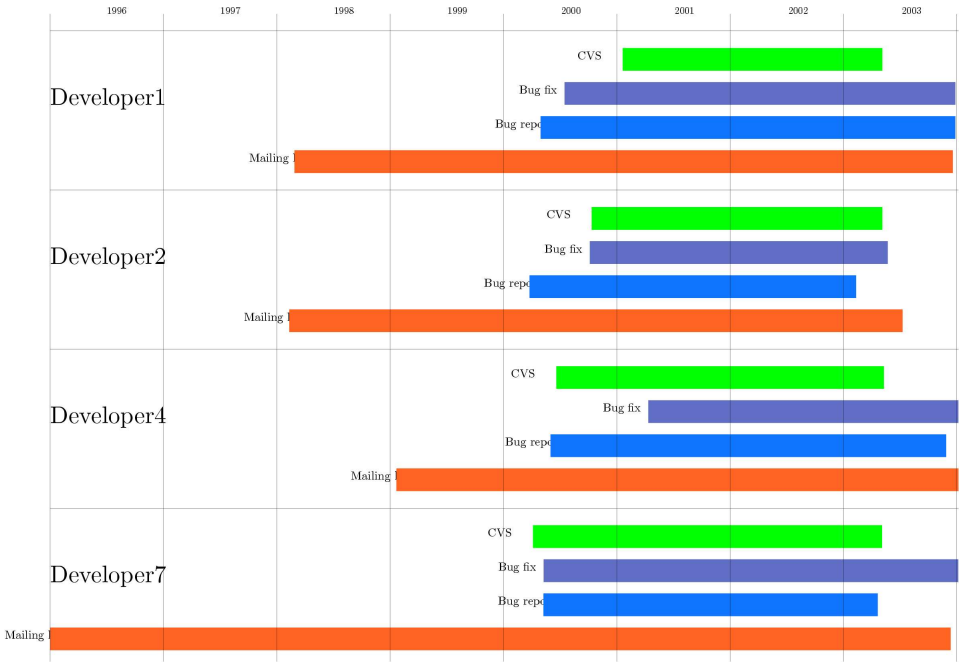


Figure 4.71: *Activity diagram for some developers in Group 1. The lower (red) line gives the time span during which the developer sent messages to the mailing lists, the next (light blue) line gives the time span for bug reports, a third (dark blue) line the span for bug fix submissions and the last (green) line corresponds to activity in the CVS repository.*

Another interesting observation is that the time from the first message to a mailing list to the first commit and the first bug report is close to 30 months. This indicates that the mean time that a member of this group needs to acquire enough knowledge of the project to begin to contribute is of about two and a half years. During this period developers become more and more integrated into the project. After this learning period, a volunteer is ready to contribute with code to the project.

4.8.6 Group 2: sudden joining

The main characteristic of the joining process in the second group is that it is much shorter: developers perform their first commit much earlier (figure 4.72 displays this behaviour visually). This indicates a

	1st message	1st report	1st fix	1st commit	commits	messages	bugs	comments
1	01/09/99	03/11/01	03/11/01	12/01/01	1603	360	158	975
2	12/23/98	02/03/01	08/16/01	08/23/01	1982	141	59	993
3	09/08/99	07/18/01	05/21/01	06/19/00	973	138	32	407
4	12/03/99	04/12/01	03/13/01	05/01/01	8044	965	154	3508
5	01/01/97	02/18/01	02/18/01	02/06/01	6949	1499	18	123
6	06/05/98	04/24/99	10/29/99	04/29/99	9485	1214	558	2080
7	11/13/96	03/20/01	03/21/01	02/15/01	3720	163	26	367

Table 4.54: *Some statistics about the selected developers (Group 1)*

faster integration process, which could be explained by developers being either hired by companies or being university staff. Supposedly, that would mean that they would be experienced developers, and therefore already knowledgeable in the uses of the coding community, and capable of learning quickly. Maybe it would also imply more confidence by their partner developers (since in GNOME, as in many other libre software projects, access to the CVS is granted once *core group* developers perceive that a candidate has enough knowledge on the project).

Figure 4.72: *Activity diagram for some developers in Group 2. The lower (red) line gives the time span during which the developer sent messages to the mailing lists, the next (light blue) line gives the time span for bug reports, a third (dark blue) line the span for bug fix submissions and the last (green) line corresponds to activity in the CVS repository.*

These developers also follow a different joining pattern, which can be easily observed in figure 4.72. Almost at the beginning of their participation (understood as their first message to a mailing list), they start to commit code to the CVS, and about a year later contribute also to Bugzilla. Even, some developers write first to CVS before sending a message to the mailing lists.

If we compare the absolute number of commits, messages, bug reports and bug comments between the members of Group 1 and Group 2, we can conclude that in general there is more activity in the second one. This may be the result of the sudden joining, as developers from Group 2 have had more time to work as part of the core group than those who belong to Group 1.

	1st message	1st report	1st fix	1st commit	commits	messages	bugs	comments
8	01/07/01	09/13/01	06/06/01	04/06/01	2884	5529	459	17
9	07/14/00	02/11/01	02/22/02	05/17/01	5877	103	15	129
10	11/08/00	04/12/01	12/11/01	04/11/01	6830	618	235	1490
11	06/06/01	01/29/01	05/25/01	01/26/01	5384	1551	62	1963
12	01/26/00	02/22/01	05/23/00	03/04/00	6789	5189	35	3062
13	10/06/99	05/26/00	04/03/00	02/16/00	6421	115	85	75
14	06/09/99	06/15/00	01/27/01	06/30/99	29862	2464	48	4149
15	02/05/01	03/27/01	02/07/01	02/05/01	11250	6561	180	1670
16	09/30/99	07/30/01	12/03/00	03/11/99	37705	165	26	1331
17	06/15/98	12/13/01	04/14/00	01/16/99	9384	838	45	1088
18	08/31/99	05/23/00	11/12/00	10/21/98	6379	709	2	16
19	12/12/98	02/01/01	11/07/00	02/17/98	32489	2799	117	6951

Table 4.55: *Some statistics about the selected developers (Group 2)*

Chapter 5

Lessons learned and Models

The most exciting phrase to hear in science, the one that heralds new discoveries, is not, "Eureka!" ("I found it!") but rather, "Hmm... that's funny..."
Isaac Asimov

So far, we have seen the type of publicly available data that exists for libre software projects. We have shown how these data can be retrieved and can be stored and in the last chapter some analysis methodologies have been presented. This chapter is devoted to see what we have learned from such analysis and what can be done with such efforts. In this regard, we have summed up a set of issues that can be inferred from our work as *lessons learned* (section 5.1). On the other hand, we show how to use the data and facts we have gathered in this thesis (or that is known from the related research) to verify and validate a development model for libre software (section 5.2).

5.1 Lessons learned

Throughout this work, we have applied several methodologies to analyze the data obtained from publicly available software development data sources. In all of the cases, the software projects that have been studied belong to applications that are considered *bazaar-driven*, large libre software. Our analysis have resulted in a knowledge over these projects that we have summed up in this section.

5.1.1 Lessons learned from the software evolution analysis

Section 4.2 has been devoted to perform *classical* empirical analysis from the software engineering world on libre software projects using the publicly available data described in this thesis. We have shown how Linux continues to exhibit a global super-linear growth pattern, as it was noticed by Godfrey and Tu five years ago [Godfrey & Tu, 2000]. However, super-linearity has become even more clear during the last five years.

At the subsystem level, the *drivers* subsystem shows to be the most important component, being itself the aggregate of many different smaller components (device drivers), usually built by different groups of developers. Most of those device drivers show a linear growth, but the number of device drivers is increasing, leading to a total super-linear growth pattern. We could speculate in this sense with a *natural* development linear growth trend for libre software projects, being super-linearity the result of adding up independent efforts by different development groups[1]. Future research should focus on bringing more insight into this question.

Applying the same study to the kernels of the BSD family, we have found that they are in general not growing super-linearly. However, before the year 2000 the growth of FreeBSD was super-linear, and the same can be said for some of its subsystems, and those of NetBSD. Except for these cases, the most predominant software growth pattern is the one that follows linearity.

[1]The reader should note that the sum of linear behaviors may give super-linearity; this is because linearity is observed only once the project has started and not before. So, the aggregation of two linear projects results linear *iff* the projects have started at the same time.

Super-linearity deserves some further thoughts: its appearance seems to be related to the inclusion of external code, to the existence of residual old code that does not need to be maintained (for instance, drivers for old devices), to a specific software architecture design (with an already fixed specification which has been widely tested, and therefore only coding has to be done), or to the allocation of work to different development teams. In this sense, further research should be performed on the linearly growing subsystems in order to find out if super-linearity is only achievable by aggregating work from several development groups working in parallel. In any case, it is not surprising that the Linux kernel shows such a pronounced super-linear growth as it features most (if not all) of the above characteristics.

With the aim of finding whether these results can be extrapolated to other (non-kernel) domains, we have also studied the growth of 18 large libre software applications. We have found that most of them show a clear linear growth pattern, in some cases after filtering out some ripples due to occasional addition or removal of large quantities of code.

Therefore we can conclude that, for the sample of large libre software projects analyzed in this thesis, growth is usually linear, with some cases of super-linear growth. Since the sample is reasonably large, and representative of a certain kind of libre software systems (stable, large in lines of code, with an active community and user base) we believe that this can be considered the common growth pattern for this kind of systems.

We can also conclude that there are no noticeable differences in the growth pattern before and after the first stable results (1.0) for most projects. From the graphs shown in section 4.2 it is not possible to differentiate the *pure* development phase (before the 1.0 release) from maintenance (after 1.0). This distinction has been historically done in the software evolution analysis of large industrial software systems, but seems not to be needed for libre software.

In any case, the studied systems show a growth rate that is higher than the smooth growth one by Turski [Turski, 1996] and in concordance with the findings of Succi et al. for GCC and Apache [Succi *et al.*, 2001]. This could mean that the fourth *law* of Software Evolution does not apply to these large libre software systems (although our analysis has some differences with a classical Lehman study, which should be further researched).

If this were the case, it could be a consequence of the particular allocation and availability of human resources in libre software projects with large surrounding communities. For instance, there are many tasks (such as testing and bug reporting) that are in fact performed outside the core group of developers, which means that external man power is actually actively collaborating to its growth. In other words, the flexible and auto-organized management of human resources usually found in those projects could be the reason of a higher growth rate than the one found in other, more rigid and planned cases.

5.1.2 Lessons learned from the evolution of software compilations

In section 4.3 we have shown the results of studying the evolution of the stable versions of Debian from the year 1998 ahead. We have been able to see the evolution of physical source lines of code, the number and size of the packages and the weight of the various programming languages in use. Additional results have been related to the number of voluntary developers working on Debian, and finally the effort, time and cost estimations using the well-known COCOMO method.

Among the most important evidences that we have found, we encounter that the stable versions seem approximately to double in number of source lines of code and packages every two years, an evolution that we think can only be possible to be maintained if the number of volunteers that enter the Debian project in the near future keeps path with this growth. That is, at least, what can be concluded from the fact that, until now, mean package size has approximately remained constant, so that the number of packages grows linearly with the number of code lines.

The size of the last version of Debian (3.1) makes us think that we are in front of one of the largest software collections in the history of humanity, if not the largest one. In order to create their 196 MSLOC, according to the COCOMO model, 50,000 person-years would be necessary and the cost would go up to around 6,800 million dollars. None of the other systems with which we have compared Debian (Red Hat, Solaris, Windows, etc.) can compete with Debian at the present time in size.

The top applications in size contained in Debian consist predominantly of low level applications

(kernel, development software, specific-purpose libraries...), although in recent releases the inclusion of end-user applications, such as Mozilla and OpenOffice.org shows a tendency towards end-user applications becoming more important over time.

As far as the programming languages are concerned, C is the most used language, although it is gradually losing weight. Scripting languages, C++ and Java are those that seem to have a higher growth in the following versions, whereas the traditional compiled languages have even inferior growth rates than C.

5.1.3 Lessons learned from the archaeological analysis

In section 4.4 we have presented an empirical application of the archaeology concept to the macroscopic study of projects maintained in versioning systems, with special attention to libre software projects. We have devised a methodology for that study, from which we have defined several indexes that can be used to summarize the development process from the point of view of aging and maintenance. We have also built some tools that automate the analysis, and have applied them to nine carefully selected libre software projects. To finish, we have discussed some limitations and complementary research lines.

One of the key findings of this analysis has been to show that the application of the methodology to the case examples has provided much insight about the maintenance efforts, and the maintainability of the corresponding projects. We have found, for instance, that around 70% of the current code base of GNU Emacs (considered in advance as legacy software) is younger than 7 years, or that Apache 1.3 is seldom maintained even being the most used WWW server worldwide nowadays.

From a more general point of view, the characterization of a project by several indexes that contribute with useful information about its age and maintainability is probably the key contribution of these ideas and may help in the decision-taking process by the development teams in libre software projects or by the management team in industrial software companies.

There are many possible future lines of research to explore this approach. First of all, better ways of visualizing the archaeological results from a macroscopic point of view should be considered. Other lines of research should focus on finding relationships among the parameters used in software evolution studies, and in correlating them with effort estimation. Another line could be the characterization of the continuous release process that is common in libre software projects, identifying its developments and maintenance parts, and relating them to other factors.

Some other quantitative approaches could also lead to interesting results from a software archaeology point of view. Among them, studying how the changes in documentation and in-line comments affect the maintenance process, or simply by repeating the same analysis on comments as it has been done on source code.

As a summary, we believe that software archaeology provides an interesting framework for digging in the past of a project, so that we can learn patterns and information relevant to infer its future.

Specially interesting are the limitations of the current approach for software archaeology, and how we can connect it with the more mature view of software evolution. Also, we foresee promising lines related to cost estimation, and present some final remarks about some information that we have gathered but not analyzed yet.

The laws of software evolution [Lehman *et al.*, 1997] have some relationship to the one of software archaeology: the former considers the total history of the project starting from the first time the software is publicly released, while the latter is only interested in what remains from the past. We have seen in several parts of this thesis that Godfrey et al. [Godfrey & Tu, 2000] studied, from this point of view, a libre software project (Linux) and found that it did not follow at least one of the software evolution laws, as its growth is super-linear.

Further research should be performed in order to see how to relate both views. In addition, following questions could be answered: does any kind of transformation exist between software evolution and software archaeology? And what additional information is necessary for such a transformation?

Software archaeology and the indexes that we have presented so far may also be used as a starting point for the estimation of future software maintenance and evolution costs. Ramil et al. [Ramil &

Lehman, 2000b; 2000a] have already performed some research on effort estimation based on change records of evolving software. The key ideas for walking that path are:

- A piece of software on which an author has worked on recently is (for him) easier to maintain than other pieces of software on which he worked on long ago.
- Maintaining software with which an author has never dealt with is more difficult than if he has already worked on it.
- Having the possibility of asking the original author of some code to be changed makes life easier than when that developer is not available anymore.

In any case, it should be pointed out that the software archaeology concept is still an immature, but very promising one. Future research will have to study it in detail and see how it can be of benefit for managing the software development process.

5.1.4 Lessons learned from the file-type based analysis

One of the most important lessons learned after the work described in section 4.5 is the possibility of using objective criteria to characterize software systems based on the relative importance of the various activities that take place during development. We have shown that taking commits as the main activity measure of the project in a versioning repository may be misleading and that the use of the concept of atomic commit which virtually groups files committed at once into a unique atomic interaction with the repository provides more realistic results.

On the other hand, we have generalized the concept of software evolution by applying it to all other file types different from traditional code and have observed how this can be used to infer some characteristics of the software system under study. In the case of KDE, we have seen that it throws a super-linear growth trend as it has been previously reported for the Linux kernel [Godfrey & Tu, 2000] and in contradiction with some of the software evolution laws [Lehman & Belady, 1985].

Another idea that has been introduced is file archeology which is a measure of how old the files are. This perspective allows to get information about the vivacity of the software system and an estimation of maintenance costs as having modules updated recently makes it more probable to have somebody in the development team who understands that part of the system. On the other hand, thanks to the archaeological study that we have performed on our case study we have been able to explain two *anomalies* that arise in the case of two of the file types that have been considered (basically for the development documentation files, devel-doc, and for translation files, i18n).

There are several limitations on the proposed analysis. The first one is related to the data source and the restrictions imposed to write access to versioning systems, specially in the case of libre software systems. Not everybody can commit to the repository, and in many cases changes, modifications and patches are centralized in a small set of developers (maybe just one), producing a *gatekeeper* effect. In industrial environments, this limitation is negligible, since these activities are performed by staff members, not supposed to have such restrictions.

Another limitation is given by the fact that our methodology is based on identifying and sorting files uniquely by their name, without taking into consideration their content. Although the set of files that cannot be sorted can be minimized to values that can be considered statistically insignificant, there are intrinsic errors in this process that could be surpassed if the file content is studied. So, for instance, we have assumed in our methodology that all files with an .xml extension contain documentation as we have observed that this is the predominant use of this extension in large libre software projects. Of course, this is not a general rule. Extending the current method with heuristics that consider the content of files should not be a difficult task and would offer better results. This type of heuristics could be enhanced with the file placement within the directory tree as usually files from the same type are grouped together into directories.

Finally, we have targeted commiters and have looked for communities and how they are built over time. We have seen that for some file types, specialization is given from the beginnings while for other file types specialization happens over time as the software project gets larger in software size

and number of contributors. These findings are specially interesting for non-hierarchical environments in the libre software world and of special interest for companies focusing on investing in such type of projects. Future investigations should answer what tasks should these companies aim in order to maximize their investment.

Further research could also focus on performing a fine-grained study only on the code files, sorting them by the programming language. This may help to identify communities working with them, and to study comparatively their evolution, possibly relating the findings from such a study with the parameters that have been observed in the current one, and to other aspects such as popularity and spread of languages within a project, or even productivity of languages.

Another interesting research line is the identification of which groups of files are regularly part of the same atomic commits. This has already been studied in literature for code files, but a more general study would make it possible to infer new knowledge on the issue. The underlying structure of a project could be studied from the point of view of files that are changed together (which usually implies a certain common knowledge by the developer). It should be noted that while (too much) coupling in source code has been usually recognized as a bad practice, this may not be the case in general.

As a final conclusion on this issue, we can say that in this section we have shown that looking at the versioning repositories for artifacts beyond source code gives much insight into a software project from the technical and management point of view. The rising importance of these elements is indicative for a promising path for future research.

5.1.5 Lessons learned from the social network analysis

Section 4.6 has been devoted to apply traditional social network analysis (SNA) techniques to information retrieved from versioning repositories from large libre software projects, such as GNOME, KDE and Apache. We have presented a novel approach to the study of libre software projects based on the quantitative and qualitative application of social networks analysis. Since most libre software projects maintain such repositories, and allow for public read-only access to them, this analysis can be repeated for many of them (although given its characteristics, probably it would be useful only for large projects, well above the hundreds of thousands of lines of code).

There are some other works applying similar techniques to libre software projects (like the one suggesting that large projects are more modular than small ones [Crowston & Howison, 2003], or some studies on the *territoriality* in libre software projects [Germán, 2004a]), but to our knowledge the kind of comprehensive analysis shown in this thesis has never been proposed as a methodology for characterizing libre software projects and their coordination structure. In addition, the use of weights for relationships that allow to differentiate between tight and weak links has been applied for the first time, to our knowledge, on a SNA performed on libre software projects. This is a major contribution to the analysis as the inequality in contributions and participation that we have observed in libre software projects skews results substantially.

Not only the main guidelines of how SNA can be used with libre software projects have been explained: a detailed methodology that can be easily automated has also been shown. It starts by downloading the required information from a versioning system, and ends with the graphs and tables shown, which can be interpreted to gain knowledge about the informal organization of the studied project. The methodology has been applied to some important and well known projects: KDE, Apache and GNOME.

Although these studies have been sketched just as case examples, some relevant results can also be extracted from them. For instance, it has been shown how all the studied networks fulfill the requirements of small worlds. This has important consequences about the characterization of these networks, since small worlds have been comprehensively studied and are well understood in many respects. It has also been found that the growth of those networks cannot be explained by random preference attachment (something that could be previously suspected), but on the contrary, that it matches well the pattern of preference attachment related to the weight (amount of shared effort) of links. Some other relevant results are the elitist behavior found in these projects with respect to the connectivity of modules and developers, indications of an over-redundancy of links and of a good

structure for the flow of knowledge, and the absence of centers of power (in terms of information flux). All these conclusions should be validated by studying more projects, and by analyzing in detail their microscope implications before being raised to the category of characteristics of libre software projects, so for now they can be considered as good lines of further research.

The main lesson learned after using social network analysis techniques for the study of some projects, is that it has a great potential to explore informal organizational patterns, and uncovering non-obvious relationships and characteristics of their underlying structure of coordination.

5.1.6 Lessons learned from the evolution of contributors

Section 4.7 studied the evolution of contributors, mostly volunteers, in libre software projects. Therefore, we have applied two analysis; first, we have analyzed the participation of the maintainers in the Debian project and second, we have studied how the group of most active developers (the *core* group) changes its composition over time for several large libre software projects.

We have therefore conducted a quantitative study of the evolution of the Debian maintainership over the last seven years. Hence, we have retrieved and analyzed publicly available data in order to find out how Debian handles the volatility of the volunteers involved in the project.

Some of the most interesting findings are:

- Both the number of Debian maintainers and the number of packages per maintainer grow over time, even if there is a trend towards having maintainer teams.
- The number of maintainers from previous releases who remain active is very high, with an estimated half-live of around 7.5 years (90 months). More than half of the maintainers from Debian 2.0 still contribute to the current release.
- Developers tend to maintain more and more packages as they are more experienced in the project.
- However, this does not mean that maintainers who have been in the project for more time maintain more packages than newer maintainers. In fact, in the latest release the highest packages per maintainer ratio is given for those who entered the project around the year 2000.

From these facts, it can be said that Debian maintainers tend to commit to the project for long periods of time. However, there is a worrisome trend in the last releases towards a higher number of packages per maintainer. This could imply scalability problems if the number of packages in the distribution increases and the project does not attract a sufficient number of developers.

Another issue on which we have focused on is what happens to those packages that were maintained by developers who left the project. Most of them are taken over by other maintainers so that we can state that a natural *regeneration* exists. Based on the data we have researched, those packages that are not adopted by other maintainers in the next release, and are therefore not present in it, are unlikely to be re-introduced in future releases.

Finally, we have also found that more experienced maintainers are responsible for packages that are installed and used regularly more often. A possible explanation for this finding is that essential packages of a distribution have been included in the early stages of the project, while newer packages tend to be add-ons or complements. The probability that newer maintainers are in charge of newer packages is higher than for maintainers that participate in the project for a longer time period. Hence, newer maintainers are responsible for packages that are less installed and used regularly less often.

In addition to the insight gained in this investigation, we have proposed a number of further studies to elaborate the findings of the present paper. In particular, team maintenance and its impact on the quality of packages would be interesting to research. It is also not clear why there is an increase in the ratio of packages per maintainer. Possible explanations are that better tools and practices lead to more efficiency or that with the success of libre software new volunteers show more motivation and commitment, but more data is needed before these explanations can be conclusive.

From a more general point of view, this study explores the behavior of volunteers in libre software projects, and provides some answers to why this kind of voluntary contributions are capable of

producing such large, mature and stable systems over time, even when the project has no means for forcing any single developer to do any task or may leave the project during important development phases. It is impossible to infer the behavior of volunteer developers just from the study of a single project, but given the size and relevance of the Debian project, at least some conclusions can be considered as hypothesis for validating in later research efforts.

One of them is the stability of volunteer work over time. The mean life of volunteers in the project is probably larger than in many software companies, which could have a clear impact on the maintenance of the software (it is likely that developers with experience on a module participate in its maintenance over longer periods of time). Another one is that volunteers tend to take over more work with the passing of time if they manage to stay in the project: in other words, they voluntarily increase their responsibilities in the project. Whether this is because it is easier for them, because of their experience, or because they devote more effort to the project is for now an open question. Yet a third one is the stability of the voluntary effort when some individuals leave the project: most of their work is taken over by other developers. Therefore, despite being completely based on volunteers, the project organizes itself rather well with respect to leavings, which is an interesting lesson about how the project can survive in the long term.

We have found that given that there are no formal ways of forcing a developer to assume a task, voluntary efforts seem to be more stable over time, and more reliable with respect to individuals leaving the project than we had expected in advance.

Besides our study of the Debian maintainers, another issue we have been concerned with has been the evolution of the *core* group of libre software projects over time. We have shown a methodology that can be applied to software projects that use versioning repositories. By dividing the total time span of the project in ten equally large time slots and observing the most contributing 20% of developers and their evolution, we have been able to obtain a graphical representation of the regeneration process that takes place (or not) in a software project. In this regard, we have classified three different behaviors: code-god projects, where the project is lead by the same group of developers throughout all its life time until now, generations, which is indicative for several groups of developers taking the lead, and finally a mixed characterization.

We have shown that most of the large libre software projects under study can be classified to follow the generation pattern. This means basically that new developers enter the projects and acquire the sufficient knowledge over time to form part of the leading group. Future research should pay attention to this behavior as it could be used to see how the program comprehension process takes place in libre software projects. Another issue that should be resolved by further investigations is the adequacy of the size of the time slot. In the current proposal, the whole time span of a project is divided into ten equally spaced slots, so that the duration of a slot depends on the *age* of the project. This makes results obtained using our methodology for several projects difficult to compare.

All in all, and although further research should also deepen in this question, some hints for the dynamic analysis of the composition of the core group can be extracted from this point of view.

5.1.7 Lessons learned from the membership integration analysis

Finally, in section 4.8, we have studied the patterns of the joining process of a certain population of GNOME core developers. Since these developers had to join an already running project, they had to understand the system before being able to contribute and therefore the joining process could be also understood as a software comprehension process.

Our initial assumption was that the integration process consists of a set of activities that start with occasional contributions, evolve to frequent participation and finish in forming part of the core group. In some sense, this can be seen as introducing a dynamic behavior to the static *onion* model (see 2.4.1).

For the population under study (selected from the GNOME project), we have learned that two clearly defined joining patterns exist: one that follows the onion model, and another one that we have called *sudden joining*. The former applies to volunteer developers in the studied sample, while the latter applies to those hired by companies for working on GNOME or for developers that belong to the staff of universities.

The first group follows a common way of approaching the project, and a learning process that lasts for about thirty months. The first contact with the project that we have tracked for this group is a message to a mailing list, followed by bug reports and fixes, and finally, active participation in the CVS repository.

The second group follows a different approach to the project. The learning process is much faster (usually less than one year), with the first commit very close to the first participation in the project (and in some cases, even the first participation being a commit in the CVS repository). In addition, there is no evidence that the predictions of our dynamic onion model can be found in their joining patterns.

Therefore, we can say that the dynamic onion model is validated only by the volunteer developers in our sample, but not by those working for the project as employees. However, for those developers validating the model, the observed joining patterns are quite similar.

Although the sample selected for the study could seem tiny at first sight, it is important to notice that it has been designed to cover one of the most interesting cases of developers joining a project. However, our selection criteria leaves outside very interesting cases, such as the developers of Ximian (now Novell), probably the company with most influence in the development of GNOME, because many of them belong to the group that was active in the first stages of the project. It would be of course worthwhile to design another sample that includes at least some of them.

All in all, we believe that the approach presented in this section gives some insight on how the process of membership integration is approached in libre software projects, and could also probably be used for traditional, proprietary ones (at least for comparison purposes).

5.2 A model based on the stigmergy concept

In the chapter devoted to related research (chapter 2), we have seen how the development of libre software has characteristics that have not been found in *classical* software engineering processes. Lack (or few) pre-defined requirements, no detailed design, lack of inter-process documentation are some of the most common cited features of this way of development [Scacchi, 2004]. In addition, libre software projects are not organized in a clear, predefined hierarchical structure, where a central authority shows the way to go. But despite all of that, we have seen that they are capable of delivering useful, mature and (in many cases) high quality pieces of software. This can only be explained by assuming that they feature self-regulation and self-organization as we have been able to study in the fourth chapter dedicated to analyzing some libre software projects from various angles (see for instance section 4.6).

In this section we are going to use all the knowledge we have acquired to model the libre software environment at large. This can be seen as a first step towards having reasonable models that make the underlying process understandable and repeatable. There is still no good model on how libre software is produced as a whole. Maybe because the inherent difficulties of dealing with many projects, each with different contexts, research effort has been focused on single, usually successful, libre software projects. In any case, this is not a new idea. The use of simulation models to understand the libre software development process can be tracked back to models (similar to the one presented here, although based on another set of assumptions) proposed by Dalle and David [Dalle & David, 2003].

The model we propose is based is based on the *stigmergy* concept, a concept used to explain how some social insects perform large-scale works. Stigmergy assumes that communication between individuals happens through stimuli caused by changes in the environment, and not by directly interchanging information. With stigmergy we are able to explain processes that act like autocatalytic reactions: a kind of behavior that we have observed in bazaar-driven self-organized libre software projects.

Based on this concept, we have designed a model for studying the evolution of the libre software landscape, and of the projects in it. The model deals with how developers are allocated to projects (or how developers decide to join projects), and how that impacts on the relative evolution of those projects. We have also implemented a system for the simulation of the model (which has been calibrated using data from studies presented in this thesis, whether found in the literature or in the previous chapter with the various analysis), and verified its output comparing its results to the results of the research on real libre software projects.

This effort can thus be seen as a first step towards understanding the social and computer-mediated interactions that yield, as a final product, a libre software project, and, as such, it can be used to improve those interactions in order to produce software products more efficiently, even in a non-libre software environment.

5.2.1 Self-organization through stigmergy

In the late 1950s the French biologist Pierre Paul Grassé realized, while studying the construction of termites nests, that some behaviors which lead to collective coordination were consequence of the effects produced in the local environment by previous behaviors (usually of other termites). He called this phenomenon stigmergy[2] [Grassé, 1959].

Grassé observed that when termites build their nest, they start randomly, without any coordination. Once a certain point of activity is reached in an areal, it becomes a significant stimulus for other termites which then start to collaborate in that same area, leading to the construction of the nest.

Stigmergy is observed mainly in social insects, such as termites, ants and some kinds of spiders. Their activity does not depend on the direct interactions with other insect-workers, but on the structure of the environment. Individual behavior is controlled and guided by previous work, i.e. the changes in the environment have a direct impact on the self-organization and coordination of the colony. An insect creates, with its activity, a structure that stimulates other members of the colony, causing them to perform other specific activities.

[2]From the Greek 'στιγμα' (mark/sign) and 'εργος' (work)

One of the ways stigmergy is observed in social insects is by means of chemical marking, depositing pheromones in specific places (for instance where food is found). Several deposits in the same place have an additive effect, causing in the end an autocatalytic reaction[3]. Stigmergy is, therefore, a coordination mechanism, characterized by the lack of *a priori* planning and explicit or direct communication between entities. Information exchange is done through changes in the environment, which usually have only local effects, and can therefore exert influence only nearby where they have been produced.

In our study we apply the stigmergy concept to libre software development. Although in general ant algorithms have been used for optimization in several contexts, our research is focused on the process itself, rather than on optimality. Our aim here is to verify whether developers have, when they develop software, a behavior comparable to that of stigmergic insects. Therefore we will create a model and test it against data that we have obtained from studies shown in this thesis and in literature. Biologists have made some efforts to model how insects interact by means of stigmergy. That is why we will first introduce with some more detail a model that simulates the behavior of ants when looking for food. In the next section, a similar model will be proposed for libre software development.

Figure 5.1[4] shows the paths followed by ants in their search for food. Ants are depicted as small points at the end of their path. The food and the nest are signaled in both parts of the figure. The picture on the left shows the way ants proceeded before finding the food. Once an ant finds food, it uses pheromones to *mark* the way to the corresponding location.

What happens in the time span between the image on the left and the one on the right can be summarized as follows: since the moment when food is found by the first ant to the moment shown in the picture on the right, the pheromone path gets stronger and stronger because of the transit of other ants (auto-catalysis), while it gets optimized by a partially random behavior (which is explained afterwards in detail). The randomness explains why even when there is a (possibly optimal) way to the food, there are still some ants which follow an alternate path. This can be observed by the existence of the other gray paths on the right picture that are different from the main path.

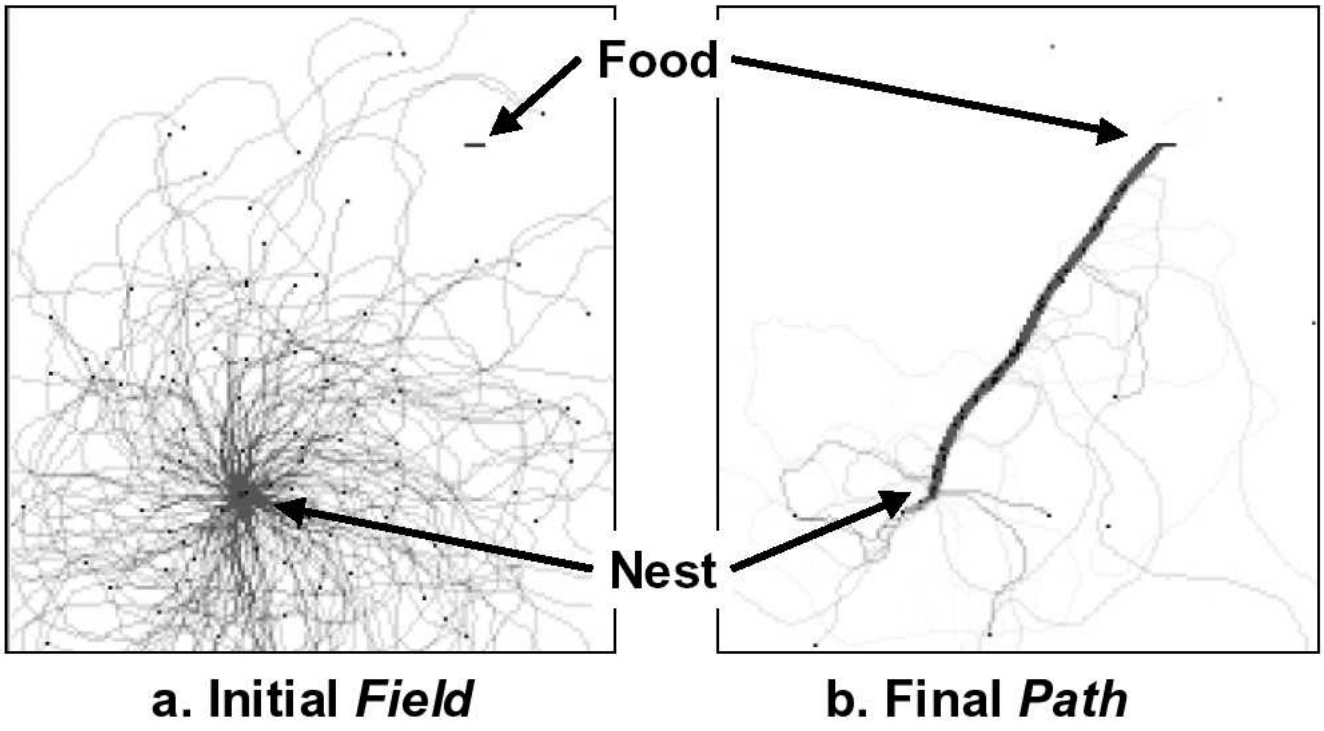


Figure 5.1: *Optimal path from the nest to the food. Ants solve this problem by using stigmergy.*

[3]In an autocatalytic reaction a product of the reaction also acts as a reactant, modifying the speed of the reaction

[4]This figure has been taken from the book "Swarm Intelligence: From Natural to Artificial Systems" by Bonabeau et al. [Bonabeau *et al.*, 1999].

So far we have seen the model at the macroscopic level. But to apply it computationally, the best strategy is to consider it at the microscopic level. Therefore, we take ants as single entities that are autonomous. The logic of every entity (in this case an ant) involved in the process is as follows: at the beginning, any ant looks in the environment for information about food, i.e. smelling pheromones that have been deposited by other ants. An ant can follow a certain pheromone mark, depending on whether a random function is below (or above) a given threshold. The threshold depends of course on the intensity of the stimulus: the more intense, the higher (or lower) the threshold value will be, and the more likely the ant will follow it.

Once the ant moves, it can find food or not. If food is found, the ant takes some of it and goes back to the nest, secreting pheromones on its way. If no food is found, the ant just moves on and gets new information from the environment. In other words, the process restarts from the beginning. The whole process can be represented as a flow chart (see figure 5.2).

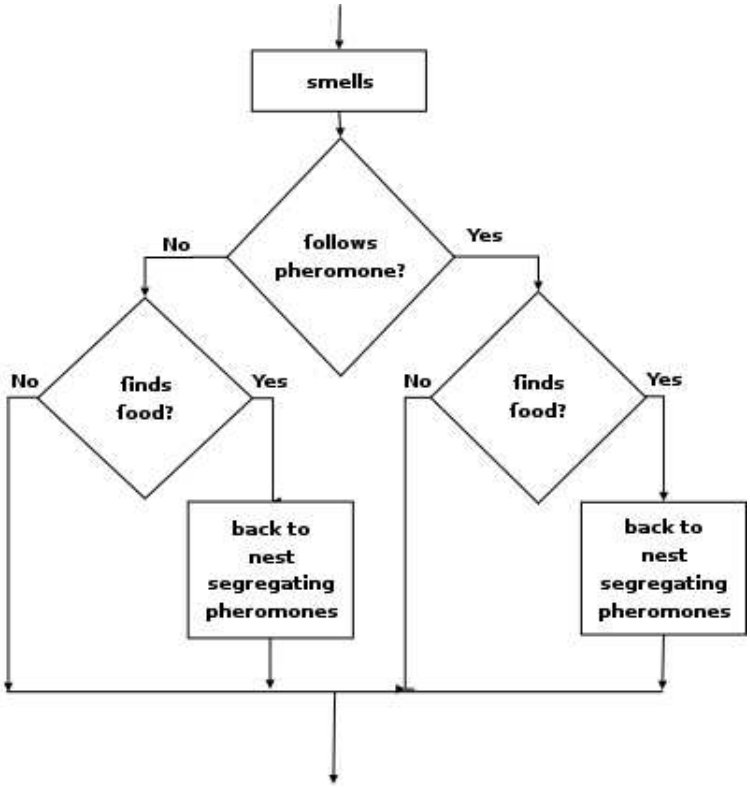


Figure 5.2: *Stigmergic algorithm used by ants to find the optimal way to food from the nest.*

This model assumes that the whole system is composed of a set of ants, but that any individual is treated as a single entity. It also assumes that the system is changing dynamically: new stimuli appear in the environment while ants find food and take it to their nest. Pheromones have also an evaporation coefficient, which causes old information to become less and less important with time.

5.2.2 Modeling libre software development

Before presenting the stigmergic model for libre software development, we introduce the basic assumption that will lead to it. We consider that developers modify the environment by producing *development activity* (i.e. source code). The production of code is the first step towards building software systems, whose presence may be seen as a stimuli for other developers to work on them. We assume that the larger a project (in source lines of code) is, the more likely it will be that a random developer collaborates on it. This is so because large projects get more attention, and may target the needs of more developers (among other reasons). So, our main idea is that source code attracts developers who create more source code, which attracts more developers. It should be noted that while source code is considered as the stimulus in our model, developers could also be considered as

stimuli by themselves as they are also attractors of other developers.

With this assumption in mind, our hypothesis is that an autocatalytic process, similar to the one found in the behavior of colonies of social insects, will take place. In other words, source code and activity around source code are in our model the *pheromones* while libre software projects are the *environment*.

The model for developers should be, in principle, more complex than the one for ants, since the latter make use of a *genetic* knowledge that is quite different in nature to the one that developers need to create software. For instance, we have to consider program comprehension aspects, such as the learning process and the acquisition of experience.

Entering into detail, our model (which has been implemented in a computer program) consists of three classes: map (Map), projects (Project) and developers (Ant). The high-level details of these classes are:

- The map is conceptualized as a two dimensional quadratic matrix of a given size, n, where each cell corresponds to a kind of software. Cells contain projects and developers. The length of the map, n, is much smaller than the number of developers, N. Any cell may contain as many projects as developers have created in it. in it. In summary, this matrix is an abstraction of a software map with several application domains.

- Projects are identified by their name, and their primary characteristic is their size (in number of lines of code) which will at the same time act as the stimulus to attract more developers. They contain a list of developers that may be active or not in the project. Projects are located in the corresponding cell, according to their application domain and hence cannot change their position. Our model assumes that the larger a project is, the more *stimulating* it will be for developers to work on it.

- Developers are the active agents in the model. They can be identified by their name, and their primary characteristics are their current location in the software map, and their technical skills and experience. Skills and experience will depend on the application domain (i.e. on a position in the map), and may change over time by using or developing software for that application domain. Developers can move from cell to cell (moving to new application domains) in the software map, but they can only participate in a project in the cell where they are located.

The dynamic part of the model is provided by developers, as described in figure 5.3. The algorithm is based on the one shown previously for social insects (see figure 5.2). The flow diagram contains several paths depend on decisions to be taken by the developer. Those decisions depend on several factors that will be presented in the following subsections. We will differentiate between the high-level abstractions of the model from the lower-level details that are required for its implementation in a computer program. The reason for doing it this way is that we do not want to have interferences in the higher abstraction level from the lower level decisions, that are sometimes arbitrary or based on partial information.

The stigmergic algorithm for libre software development

The proposed algorithm works as follows: developers work in turns; the first action in every turn is to determine the position in the map where the developer will be located. Once that position is known for the current turn, whether the developer finds (gets to know) a project in it will be checked.

If the developer finds an already existing project, he may collaborate on it, learning (gaining experience) and improving his skills. The amount of work performed will depend both on his previous experience and skills, as well as on the time devoted to it. But if the developer, even finding a project, decides not to work on it, he may still learn from it by using the software. In any case, learning by using happens at a slower rate than by collaborating in the production of code. On the other hand, if the developer does not find a project, he may decide to create a new one.

All these behaviors are modeled as probabilistic functions, in which a random value has to be above a threshold that may depend on parameters carefully calibrated. Thresholds are normalized

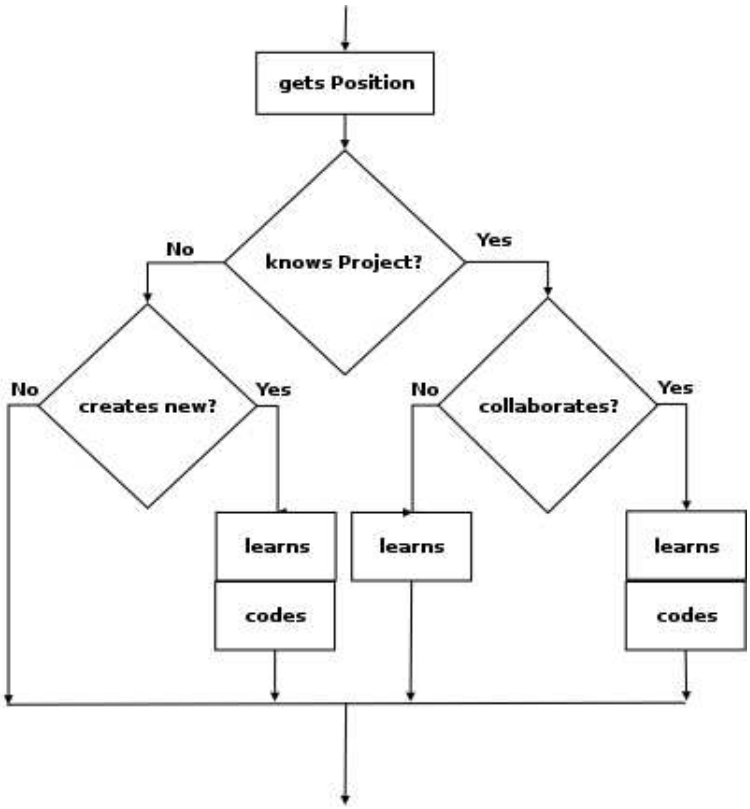


Figure 5.3: *Stigmergic algorithm used to model libre software developers in each turn.*

to one and may have maximum/minimum values to avoid that a path becomes impossible to follow. Further details are presented in the next subsections.

5.2.3 High level abstraction

In this subsection, we take all the blocks of the diagram shown in figure 5.3 and show the mathematical expressions that drive the behavior of the agents. We will maintain a high level of abstraction by just considering the factors that influence each expression leaving for the next subsection the discussion and determination of the relationships among the factors and the values of the constants.

Getting a (new) position in the map

The developer gets at first a random position in the software map. This position may change at the beginning of every turn. This is done following a probabilistic function that depends on several parameters. We have considered that there exists a natural dullness to remain on the same position, given by a minimum value for the threshold. In addition, there should be a maximum possible value for the threshold, so that there is always a chance of moving to other places in the software map.

$$threshold = f(dullness, skills, experience) \tag{5.1}$$

Hence a developer moves to a new (random) position if:

$$random\_value(0, 1) > threshold \tag{5.2}$$

Finding a project

Once a developer is on a given position, he will look for projects. Some parameters will make finding a project more probable: the project size (in lines of code), the number of developers participating and the time that the project exists (age). Being large in size, having many contributors and being

old makes a project more easy to find, up to a minimum threshold value that assures no project is known by anyone anytime. So, we will assume that the threshold depends on an initial value that will give the probability of finding a project. We have also introduced a minimum value of probability, so that there will never exist total certainty that a project is known in advance. The resulting formula is:

$$threshold = f(size, developers, age) \tag{5.3}$$

Hence a developer knows (finds) a project if:

$$random\_value(0, 1) > threshold \tag{5.4}$$

Participating in a project

The skills and the experience a developer has on a position are going to be the parameters that affect the calculation of the threshold value. Again we will have a maximum value that will make it possible that even without skills and experience a developer works on a project, as well as a minimum value which implies that working on a project may not happen even having high values of skills and experience. The minimum value could be selected proportionally to vacation time or similar. Hence, we will have a relationship like this:

$$threshold = f(skills, experience) \tag{5.5}$$

Thus, a developer will collaborate on a project if:

$$random\_value(0, 1) > threshold \tag{5.6}$$

Creating a project

In this case we assume that the developer has a natural inclination to create a new project, which depending on the implementation details may be high or low. The natural inclination will be affected by skills and experience on other places in the software map, specially if they are located near the current position. The threshold for creating a new project will depend on:

$$threshold = f(inclination, skills, experience) \tag{5.7}$$

In our model a developer creates new project if:

$$random\_value(0, 1) > threshold \tag{5.8}$$

Amount of work (code) performed

A developer acquires experience by working on a project. Experience is measured in terms of source lines of code and depends on the application domain (i.e. the position in the software map). By coding a developer produces lines of code for a project. The amount of work that is done depends on the time that the developer has committed to this task and on the skills and experience of the developer in that position (i.e., in the software domain). The amount of work (measured in lines of source code, SLOC) performed by a developer so far on a given position will be used to have a measure of his experience on that software domain.

$$SLOC = f(skills, experience, time) \tag{5.9}$$

Learning rate

Learning can be done by submitting code to a project, or just by *using* a project. *Using* should be understood in the advanced user way, so that reading the mail archives, submitting bugs and reading technical documentation is comprised. By learning a developer acquires skills.

$$skills = f(using\ or\ developing) \tag{5.10}$$

The function for skills and the one for experience should not be simply additive, but should take also into account that as time passes both parameters decrease.

5.2.4 Implementation details

The parameters and thresholds that have been used in the model have to be configured properly in order to obtain a realistic model. Data from previous research studies, and surveys performed on libre software developers have been used for this purpose. So, for instance, the mean time spent on a project by libre software developers has been taken from several surveys, and productivity has been measured in SLOC, in order to be comparable with the application of the COCOMO cost estimation model [Boehm, 1981].

The stigmergy model presented so far has been implemented with a Python-based program called pyStigmergy[5]. Hence, the previous high level abstraction formulas had to be specified and implemented. The main implementation details are explained and discussed in this subsection. We will follow therefore the same procedure as in the previous subsection having some paragraphs for each block in figure 5.3.

Getting a (new) position in the map

The natural dullness to remain on the same position (used as a minimum value) has been set to 0.7, while the maximum has been given a value of 0.99. Choosing 0.99 means that if turns are weekly, developers will (in the mean) change their position every two years.

$$random\_value(0,1) > 0.7 + \frac{skills}{100} + \frac{experience}{10000} \tag{5.11}$$

The normalization values have been put to 100 for skills and 10000 for experience. More insight about these values (and their normalization) will be given below.

Finding a project

The maximum value of the threshold for finding a project has been set to 0.8, which in fact means that finding a project is by default difficult. The minimum value has been set to 0.01.

Size, number of developers and age have been normalized into formulas to give numbers that range from 0 to circa 6. All formulas are monotonic increasing and have the property that new code/developers included when the project is small affects the visibility more than when the project is already large. In other words; adding a new developer to a small project will have more impact than doing it on a larger one.

$$size = \frac{ln(project\_size)}{2} \tag{5.12}$$

$$numDev = \frac{\sqrt{project\_numDev}}{2} \tag{5.13}$$

$$age = project\_age * 0.05 \tag{5.14}$$

[5]The program is libre software, licensed under the GNU General Public License (GNU GPL), and can be downloaded from http://libresoft.urjc.es.

Age has been taken to be linear and measured in weeks with a factor of 0.05. Age is given by the number of turns a project has been worked on by at least one developer. This way, age can be seen as a proxy for a release, so the model includes the 'release often' paradigm as stated by Raymond [Raymond, 1998].

As we can see from the next equation, the three parameters are treated equally by summing them up and by dividing them by a value that is dependent on the constant. The 64 has been taken as a normalization value and is the result of 4^3, which is a sort of mean of all parameters.

$$random\_value(0,1) > 0.8 - \frac{size + numDev + age}{(1-0.8) * 64} \tag{5.15}$$

Participating in a project

We suppose that the maximum value is given by a constant that has been set to 0.5, meaning that a newcomer has the same probability of working on the project than not doing it. Skills and experience (properly weighted in order to become normalized) will lower the threshold up to a minimum value of 0.01.

$$random\_value(0,1) > 0.5 - \frac{skills}{1000} + \frac{experience}{50000} \tag{5.16}$$

Creating a project

We assume that developers are not very inclined to create a new project, so we have put a constant threshold value of 0.8 that should be surpassed. Although the conceptual model discusses other parameters, we have not considered them for our implementation.

$$random\_value(0,1) > 0.8 \tag{5.17}$$

Amount of work (code) performed

The amount of work performed depends on the time devoted to a project. Based on the FLOSS survey [Ghosh *et al.*, 2002a] which was answered by over 2500 respondents, we have chosen a probabilistic function that descends linearly from 1 hour to 40 hours (see figure 5.4). The maximum involvement is 40 hours/week. Involvement is calculated randomly from 0 to 1 and then transformed to the previous probability function by means of the following equation:

$$time = 40 - \sqrt{1600 - 1600 * involvement} \tag{5.18}$$

The production of the developer depends on its skills and experience in this position. The maximum value for skills * experience is 50.

Finally, the work done (measured in SLOC) is given by the multiplication of all the parameters. As skills, experience and time have upper bounds, the maximum amount of SLOC that can be produced by a developer in a single week has also an upper limit: 2000 SLOC[6].

$$worked = round(skills * experience * time + 0.5) \tag{5.19}$$

Creating a project works differently. We have supposed that a random amount of code is added at first because of the programming structure and an initial size that the project has to have before being released (this can also be understood as the "fitness factor" introduced to explain why some web sites have more links despite their recent creation, the young *start-up* phenomenon, as discussed in 2.4.2):

$$worked = rand(0,1) * 800 \tag{5.20}$$

[6]Although this is the maximum value that may be obtained only under certain, very specific circumstances, it appears to be very large (compared for instance to the estimation given by the COCOMO model [Boehm, 1981]). In section 5.2.5 we will discuss these parameters and their significance in detail.

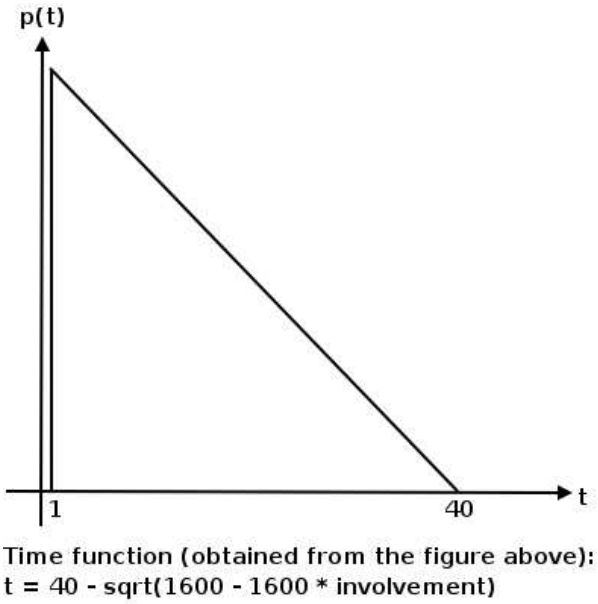


Figure 5.4: *Time distribution.*

This means that we assume that contributing a new project has in mean 400 SLOC more than a contribution to an existing one.

Learning rate

If a developer works (submits code) to or uses a project, his skills get increased following this equation:

$$skills = skills + factor * rand(0, 1) \tag{5.21}$$

where the value of the factor is 2 for developing and 0.5 for *using* the software.

Other implementation details

Our implementation assumes that the matrix is quadratic and that its size does not change over time. In order to see how the size of the matrix affects the results, we have run our simulation for several values that range from 100 cells (10x10) to 2500 cells (50x50).

The number of developers is another dimension which we have to set, specially as the relation matrix size/number of developers will give the concentration of developers and this will probably affect the results. Thus, we simulated with values for number of developers that go from 50 to 10,000. The number of turns (weeks) was set to 750, around 15 years.

5.2.5 Validating and verifying the model

The verification of the model has been performed against results obtained from several research studies. We can classify these results in two sets: facts about developer contributions, and evidences about the software produced. In both cases, with the former calibration, our model shows similar trends to those obtained in those studies.

Regarding developer contributions we have observed whether the model shows a mean involvement, as reported by Hertel [Hertel *et al.*, 2003]; inequality patterns for developer contribution to projects, as found by Koch et al. [Koch & Schneider, 2002]; whether projects are led throughout the lifetime of the project by several generations of core groups (see subsection 4.7.9); and whether the number of developers in projects follows a power-law distribution [Healy & Schussman, 2003]. Other contribution patterns, as stated by Mockus et al. on Apache and Mozilla [Mockus *et al.*, 2002], have also been compared with the results we have obtained.

Regarding the size of the developed software in the model, we will check whether the average size of software projects remains constant over time, and if we obtain a power-law with project sizes as some studies on GNU/Linux distributions have thrown (as observed in section 4.3). Finally, we will look at the software evolution patterns that our model provides and compare them to the current state of the art (see section 4.2 and [Godfrey & Tu, 2000; Lehman *et al.*, 1997]).

Mean time per week

Hertel et al. [Hertel *et al.*, 2003] report from a survey made to 141 Linux kernel developers that the mean time these developers devote to developing libre software is in the mean 12.37 hours per week.

For all the values for the matrix size and the number of developers with which we want to validate our model (in total, we have used 9 different combinations of map sizes and number of developers), the mean time devoted to development laid between 11.4 hours/week and 11.9 hours/week. The statistical mean for the probability function of time is 11.71 hours/week. In other words, the results given by our model seem to be near the ones known from literature.

Number of developers and age of a project

Krishnamurthy [Krishnamurthy, 2002] made an analysis of around 100 mature libre software projects. He stated that the number of developers working on a project was correlated to the age of the project. An analysis of our simulation shows that this is a common trend.

Distribution of work

Mockus et. al [Mockus *et al.*, 2002] reported that certain libre software projects as Apache and Mozilla rely on a small group of developers (called the *core*) who control the code base. This group is responsible for around 80% of the contributions. Other research studies have shown this trend for other projects, as for instance Koch for GNOME [Koch & Schneider, 2002]. From a general point of view, Ghosh et al. found that this happens to be in general for all libre software [Ghosh *et al.*, 2002b; Ghosh & Prakash, 2000].

Developers	Map Size	Mean	2nd Quintile	4th Quintile
50	100	0.74	0.71	0.91
50	2500	0.53	0.44	0.82
75	100	0.73	0.71	0.91
75	2500	0.50	0.40	0.80
100	100	0.75	0.72	0.91
100	625	0.62	0.56	0.86
100	2500	0.5	0.41	0.81
500	100	0.70	0.69	0.91
5000	100	0.69	0.64	0.88

Table 5.1: *Gini Coefficient for the projects*

A way of measuring the inequality in contributions is by using a classical index used in economics for wealth distribution: the Gini coefficient [Gini, 1936] which is tightly related to the Pareto distribution. Table 5.1 gives the results from our simulation for the various parameters we have used (*Developers* being the number of developers and *Map Size* the number of cells). The mean of Gini coefficients goes from 0.5 for a setting where there is a low developer concentration to values around 0.75 similar to the ones reported for SourceForge projects[7]. The table also shows the values for the second and fourth quintile to give more insight about the distribution of data. It can be observed that the distribution is always skewed to values nearer to 1 (high inequality).

[7]For more information about the Gini coefficient of SourceForge projects, visit http://libresoft.urjc.es/libresoft/25.

Power-law of contributors

The distribution of developers among projects has been reported to follow a power-law in a study performed on more than 50,000 projects [Healy & Schussman, 2003].

Figure 5.5 is the result of plotting the results from our model for several runs (the ones done with 100 cells). On the vertical axis we can see the number of developers (in log scale) while the horizontal axis (also in log scale) gives the projects.

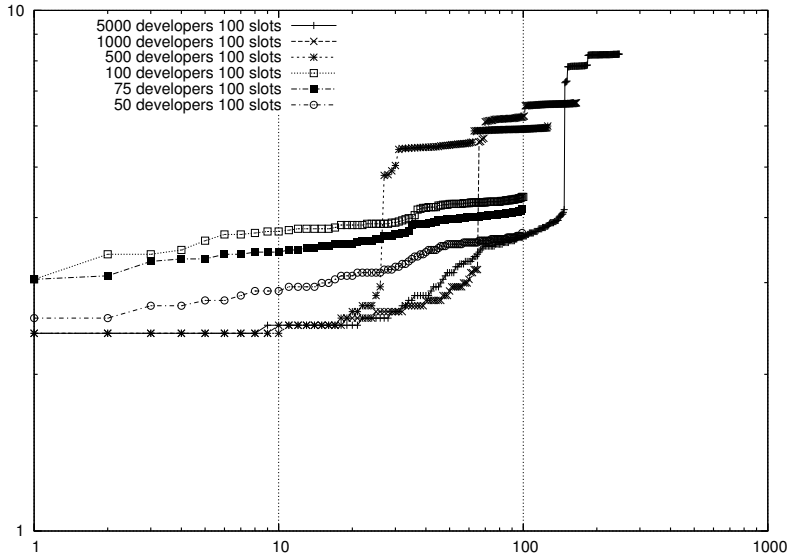


Figure 5.5: *Number of developers in log-log scale.*

The figure shows that for all curves have two different zones, both with the shape of a power-law (linear in log-log scale). Interesting is the existence of a breaking point between them, meaning that there exists a vacuum of projects with an intermediate number of developers in our model. Once projects achieve certain size, they accelerate themselves enormously. This, of course, does not comply to any study performed on libre software. It seems that this effect in the simulation depends on the concentration of developers, so probably letting the size of the software map grow over time may make this curve grow smoothly as a unique power-law.

Power-law of project sizes

González-Barahona et al. demonstrated from a study on a GNU/Linux distribution that the sizes of the projects follow a power-law curve [González-Barahona *et al.*, 2001].

Figure 5.6 plots projects in the horizontal (logarithmic) axis and project sizes in the vertical (logarithmic) axis. The resulting curve is clearly power-law for all the cases plotted with a distorting side-effect at the end of the curve possibly due to the same issues as discussed for the previous parameter (again, concentration may be too high).

Mean size remains constant

Another ecology study performed on GNU/Linux distributions over time threw as result that the mean size of the packages remained constant over time (as we have seen in section 4.3).

We have not observed in the simulations that the mean size of the projects remain constant over time for any of the parameters considered. In our model, what remains constant for all the cases considered is surprisingly the median. But the mean size has a tendency to grow even if new projects are created by developers. This could imply that the creation factor is too high and too few projects are created in our simulation or that having an existing project in a cell makes it very difficult to have other projects there too. Hence, the model should be recalibrated to fit this issue.

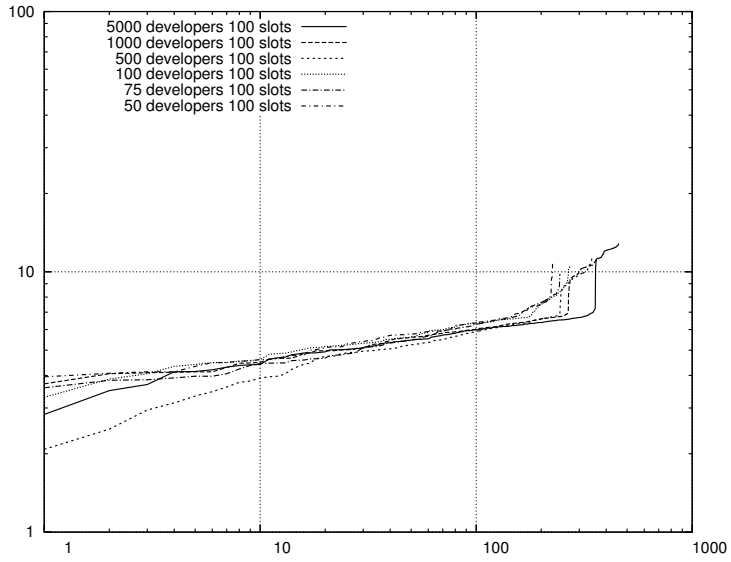


Figure 5.6: *Size of projects in log-log scale.*

Initial size

We also wanted to check if the assumption that the first contribution should be statistically larger in terms of SLOC is meaningful or not. Therefore we looked at the sizes of packages for the Debian GNU/Linux distribution over time [González-Barahona *et al.*, 2004]. In Debian 2.0 168 out of 1096 (15%), in Debian 2.1 217 out of 1551 (14%), in Debian 2.2 446 out of 2611 (17%) and in Debian 3.0 739 out of 4579 (16%) are under our mean initial value (400 SLOC), which makes our assumption hard to stay.

But looking at the model, the first quintile gives for several runs with different parameters 295 SLOC, 290 SLOC, 282 SLOC, 283 SLOC. These results validate the model.

Generations

In section 4.7 we have seen that the core group of a libre software project is not stationary over time; *generations* of core groups can be observed. In other words, the ones who started a project do not have to be part of it all the time, and newcomers take it over and make the project evolve.

Developers	Map	# Projects	% Projects	% Founder	% 1st Third
50	100	109	37.46%	9.6%	49.22%
50	625	255	20.88%	40.02%	54.44%
50	2500	5	0.22%	49.92%	54.81%
75	100	111	38.01%	5.95%	54.68
75	625	492	34.72%	33.96%	51.78%
75	2500	23	0.77%	57.76%	59.52%
100	100	118	39.6%	4.95%	54.43%
100	625	559	36.68%	27.58%	49.1%
100	2500	97	2.69%	44.83%	54.9%

Table 5.2: *Looking for generations in the core group. Contribution of the founder and oldest third of developers to the project.*

We have taken projects in our model that have a number of developers larger than 6 (see column % Projects in table 5.2 to see how many projects are affected by this choice) and have seen the percentage of the code that has been contributed by the project founder (% Founder) and by the oldest contributors (% 1st third). The mean values that are shown in the table reflect that the proportion of work made by the founder and the oldest third is in fact important, but by far not

predominant. It should be reminded that a stable core group from the beginnings would give values above 80% as discussed in previous subsections.

Software growth

There have been some studies that have investigated the system growth in libre software applications, being specially significant the one performed by Godfrey and Tu on the Linux kernel [Godfrey & Tu, 2000].

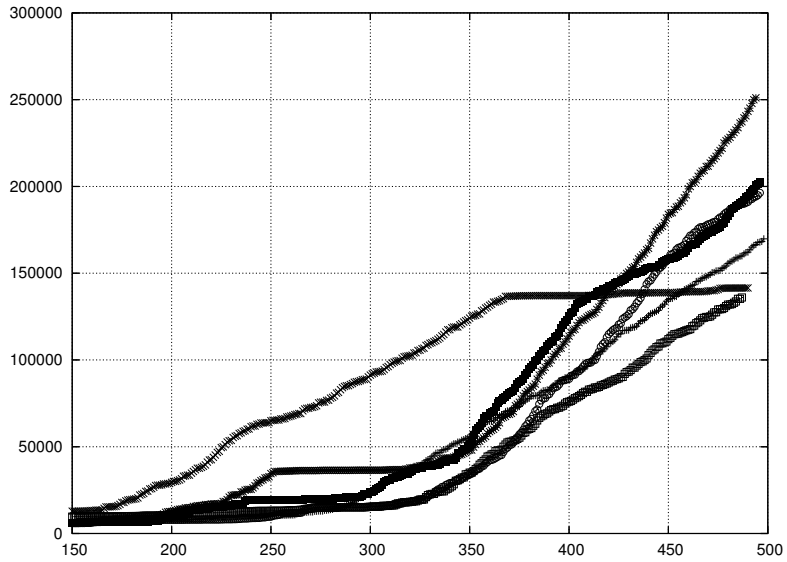


Figure 5.7: *Growth in number of SLOC for some large projects in our model.*

Figure 5.7 shows that for the largest projects from one of our analysis, no such super-liner growth can be identified. At most, projects show a linear rate with some trends of super-linearity for some time. This result is in concordance with the one observed for the majority of libre software applications that have been studied from the software evolution point of view in section 4.2.

5.2.6 Discussion

After presenting our model, we will discuss in this section some of its possible uses, its limits, and some other general considerations about its implications in the software engineering field. This will include some final remarks about the model, its limitations and applicability to other development domains and finally the discussion of some general thoughts that address key issues from software engineering.

Development attracts developers

The effect that is in the core of our model could be summarized as *development attracts developers*. This is by no means a strange factor of social interactions: activity is usually an attractor for more activity. This helps to explain from the growth of cities to the success (or not) of cafeterias in a given area. In our case, the attraction by successful libre software projects is clear. Many developers prefer to collaborate in a mature, large and well known project than in a small, hardly known, obscure one. This is reasonable from many points of view. It is more likely that a developer gets knowledge about a well-known project, that his expectations of becoming himself known are higher, or that he has some kind of relationship with a current developer (since there are more of them). On the other hand, getting help from peers is more likely, and expectations about success (joining a project which is already a success) are also higher.

Probably this is also the reason for the high rate of abandoned or one-developer projects. If a project is small, it is going to have a hard time attracting new developers, which means that no new resources are available for development, that growth is slow, that visibility will also be small. That,

in turn, makes the project not more attractive to developers... On the contrary, if a project reaches a certain size, more and more people know about it, and will eventually consider joining it. This is perfectly consistent with the power-law distribution found in the size of libre software projects and the rich-get-richer effect (see section 2.4.2), for instance.

Limitations

One of the first questions that arises when presenting a new model is under what conditions it fails to be reasonable or plausible. In this regard, we have to highlight that the main drawback of our model is that it considers no direct communication mechanisms between developers. This kind of communication, of course, exists in real libre software projects, and is in fact quite important. However, it is also important to notice that the model matches quite well real data. This could mean, from another point of view, that maybe direct communication channels have little influence on how developers choose and contribute to projects.

Attending to other limitations of the model, it is convenient to notice that there are many factors, very important in the low-level understanding of a project, which are not considered. Some examples are those related to the *external relations* of the project: documentation, blogs, support sites, web sites, etc. All of them have a clear impact on the visibility of the project, and therefore on the attraction of new developers. However, they are not considered in the model.

Another limitation that our model has is that we consider too homogeneous developers. In this sense we only take into account skills for characterizing developers and hope that the statistical considerations introduced in the model will make heterogeneity arise among developers and the model more realistic. However, some other factors such as geography or cultural affinities could also be important for the selection of a project. Of course, the barrier of entry for a project will be different depending on these factors. For instance, it seems reasonable that if there is a large group of developers of a certain project in a geographical area, new developers in that area are more likely to join that project. Or that (natural) language could impose barriers for entering a project. However, we do not know about studies which show whether this actually happens or not. Our model is global in that sense: any developer with a certain set of skills is equally likely to join a given project, with independence of where he lives, which languages he speaks, or any other peculiarity he may have.

The model assumes that the pool of available developers in a certain domain is global and thus developers seem not to be scarce resource. If there are too few of them, no stigmergic process is possible. We have not studied that lower bound, but it will probably be above the usual number of developers in most software companies. This could be the reason why stigmergic processes do not happen in proprietary in-house software environments.

Applicability to proprietary domains

Software development goes beyond libre software and as such it would be interesting to study whether this model is applicable to non-libre (proprietary) software development environments.

In recent years, some large software companies have started to look at the libre software phenomenon and to adapt their strategies to it even if not releasing their software under a libre software license. Their objective is to leverage the libre software development process as described here (autocatalytic process, to benefit from external contributions, among others) without losing the control of the project that copyright law grants them[8]. It is difficult to know the impact of the license used by a software project on the willingness of new developers to join in. In any case, licenses have different implications. The consequences of the license choice could be interpreted as positive or negative depending on the developer. We could introduce this effect easily in our model by including another factor that lowers/raises the threshold value depending on the licenses developers usually prefer and study how it impacts the total picture and, in particular, projects with those licenses that are more restrictive.

[8]This is the case, for instance, of Microsoft's Shared Source Initiative or the Java Community created around Java by SUN.

On the other hand, since our model emphasizes the availability of the source code, and in fact considers it as an attraction factor, the case of proprietary software is basically not considered at all, at least for traditional in-house developments in companies. In this sense, the model may be helpful to explain how *development communities* work in business environments. Although these communities are usually built upon partnership contracts among companies, many similarities could be observed with a libre software environment.

But some parts of the model could be used to understand and even to estimate whether it could be convenient to distribute some system as libre software when a company wants to quickly promote a large development community around it. Besides the explicit consideration that the software company should try to attract developers to enter an autocatalytic reaction, other questions could be simulated with this model. In this regard, many companies that want to invest in a software developed by the community do not know if the effort they should devote to the project is by hiring developers from this community or to contract new developers that should integrate themselves into the project. The former developers provide probably over system-wide knowledge and have already experience and expertise on the project, while the latter add new developers to the project (which makes the project more *attractive* in our model) although they require over a software comprehension phase to gain understanding over the system and become productive.

Stigmergy, the bazaar and the cathedral

If we consider our stigmergy model in the context of the bazaar, it shows some common properties. The model behaves in a certain way as a bazaar, with no formal rules, no formal hierarchy, and no task-imposition mechanisms (in our case, what project a developer joins). However, our model is more appropriate than the bazaar model when comparing it to the cathedral development model. This is because although the bazaar model explains concisely the exchange of code and knowledge in a non-controlled environment, its analogy lacks of a very important element: a final (tangible) artifact produced by all those interactions as we have in the case of software (its source code). This is not the case for the cathedral model as at the end of the process we have, obviously, a cathedral.

And that is precisely the reason why we think our model is a better analogy for the development in libre software environments as stigmergy may also lead social insects to build complex structures. This is the case for termite nests which could in fact be considered literally as cathedrals made of clay. This means that with the stigmergic model we have a complete analogy that allows to compare the construction of cathedrals as it was done by humans several centuries ago (following processes similar to traditional in-house software development) with the construction of termite nests (which obeys to behaviors that can be found in the development of libre software systems). In addition, our model illustrates that developers need to be stimulated in order to make such a development process happen while the bazaar analogy does not consider how the bazaar comes to be and does not explain the heterogeneity present in the libre software landscape.

Information hiding

Finally, we have to enter into a historical debate about how information should be managed in software development processes: the classical Brooks [Brooks, 2001] vs. Parnas [Parnas, 1972] argument about information hiding. While Brooks argued for having public access to information, Parnas proposed in the early seventies that there should be some design and implementation decisions that software developers should hide in one place from the rest of the program.

Our model is certainly more oriented towards Brooks' position. Information (source code) should not be hidden as it is the stimulus that makes an autocatalytic reaction possible and the whole software development phenomenon successful. Interestingly enough, in the 20th anniversary edition of the "Mythical Man-Month" Brooks states that this was one of the seldom errors in the original version [Brooks, 2001] . Probably from pure technical point of view, information hiding is more appropriate and is a practice that should be followed. But, as we can see from the model, software engineering sometimes has to consider other factors as development does not happen in a vacuum. Software projects that intend to be based on stigmergy should manage information hiding in a balanced

way to assure software quality and developer attraction at the same time.

5.2.7 Summing up

This model is a first step towards understanding the social and computer-mediated interactions that yield, as a final product, a libre software project. We have therefore proposed and verified an analogy with how social insects perform large-scale works by means of indirect communication entities that act as stimulants.

Our primary conclusion is that libre software development can indeed be modeled as a stigmergic one, at least with respect to how effort is allocated to projects, and to how this affects the evolution of projects. The model we have proposed shows patterns similar to those reported in real world studies on libre software, although finer calibration and further discussion on the variables and threshold values is necessary.

An interesting conclusion that can be drawn from this model is that the individual productivity may not be as important as the total production as a community; or that stigmergic mechanisms could be used in order to increase productivity at the project level. On the other hand, our model raises a set of interesting questions the we have briefly discussed. We have seen that development attracts developers and that this activates a rich-get-richer effect which makes successful projects even more successful and attractive to new developers. We have discussed the limitations that our model exhibits and have presented our doubts about if a stigmergic process could be applicable in proprietary environments because of the lack of critical mass and the fact that developers may have different threshold values for more restrictive licenses. Finally, we have indicated how stigmergy gives a clearer analogy of some of the factors that characterize libre software development than analogies that are commonly found in literature (mainly, the bazaar). We have finally discussed how our model fits into the classical information hiding debate.

Future research should focus on studying how the modification of some of the parameters may affect the development of software in the libre software world. This could be the case of a company wanting to hire developers for a libre software project. An interesting extension could be to study how other *communities* present in libre software development (i.e. translators, documenters, among others) can be included in the model. And, of course, having a software program that implements the model visually will help in the better understanding of the whole process.

Chapter 6

Conclusions

> *Einstein argued that there must be simplified explanations of nature, because God is not capricious or arbitrary. No such faith comforts the software engineer.*
> Frederick P. Brooks Jr.

In this thesis we have seen how to study publicly available information from a software engineering perspective, especially attending to information obtained from libre software projects. We have started on a basis that this type of projects offer a high quantity of data that would help us understanding their software development processes. Then, we have presented related research (chapter 2) regarding others topics like the development of libre software and mining software repositories and we have provided a detailed study of the data sources, that goes from the identification, the retrieval, the data extraction and the storage of the various data sources to its analysis (chapter 3). The part devoted to the analysis of the data has been supported with several methodologies that identify technical and socio-technical characteristics of the software projects under study (chapter 4). Based on this we have extracted some lessons learned about the projects used as case studies and have proposed a macroscopic model that explains, at least in part, the development of libre software (chapter 5). The model has been adjusted and verified with *real life* data from libre software projects or with findings from related research presented in this thesis.

Finally, this last chapter is devoted to present the main conclusions of this thesis, together with other contributions. Some of the limitations of the research methodology presented will also be discussed. The last section is devoted to some hints about future possibilities in this research area.

6.1 Main contributions

This thesis offers two major contributions from the point of view of software engineering research. The first one is concerned with the use of public data sources as knowledge generator for the practice of software engineering, while the second one states that this is the first time that an exhaustive analysis of many libre software projects has been performed. Besides the main contributions, which are high-level contributions, a set of other more detailed contributions will be summarized in section 6.2.

6.1.1 Public data sources as software engineering knowledge generators

This thesis has shown that publicly available data sources are an important knowledge generator about software projects and that they provide help during its development, maintenance and evolution processes. We have seen that we can infer in a non-intrusive form new data and facts of the whole process that can be used for technical and socio-technical purposes. By considering the various data sources that libre software projects provide, researchers and managers have the possibility to acquire dispersed knowledge on the software project in an almost-automatic way.

In this sense, we have presented methodological approach to the data sources that ranges from its retrieval to the analysis and integration. Based on this methodology, tools have been designed, built and tested against existing libre software projects to ascertain the validity of this approach. A large part of this thesis has been concerned with the data and facts that we can infer from the tools

and how to avoid inconveniences and disadvantages observed while applying them to libre software projects. The acquired knowledge will have to be supported in future with further experiments on more software projects to gain more experience with the methodologies and tools.

6.1.2 Exhaustive analysis of libre software projects

The second major contribution of this thesis is the exhaustive analysis of a high amount of (libre) software projects. In this thesis, we have worked with over two dozens of large libre software projects applying our methodologies and tools. The projects, usually large in size and in number of contributors, have been used as case studies for the various approaches that have been presented and give hence consistence to the research goals of this thesis. We can conclude that it is feasible to extract data and analyze multi-million software projects with thousands of developers, including data from various sources and in several points over time.

But beyond this, the methodologies have been thought to be applicable in general to any other software development project. If the project to be studied makes use of the most popular development tools in the libre software world, then the tools that have been built by third parties (and presented in this thesis) or by the author of this thesis and the research group he belongs to can be used.

6.2 Other, specific contributions

Besides the major contributions that reflect high-level concepts and goals, some other, more specific contributions have been made in this thesis. These contributions range from *pure* software engineering analyses to others that have a social and resource management point of view. The range of ideas is widen by our model, which uses an analogy from the biological world of the social insects to explain the software development process at the large.

6.2.1 Identification, detailed description and integration of data obtained from development-supporting tools

One of the more specific achievements of this thesis has been to depict in detail all the major telematic tools that are being used in the development of libre software projects. As such, we have had a detailed overview of the source code, the mailing list messages, the versioning system commits, the bug reports and other, project-specific data (in our case, especifically from the Debian project). Besides the identification of what data each data source may offer that is worth researching, we have offered a new way of integrating data obtained from various sources. This is an idea on which other research groups have already been working, especially by identifying traces of artifacts in different sources or by detecting references that link to other sources. Our approach is slightly different. Our intention is to link those activities that have been performed by the same person. In general, developers have different forms to identify themselves in the development-supporting tools such as a username, an e-mail address or their real name. The identification method they use will depend on the tool that is used (for instance, versioning systems require to have a username, while for mailing lists and bug-tracking system the e-mail address is used).

Managing information with personal data, as we are doing, implies to face some concerns with the privacy of the developers. Our proposal has therefore been thought to allow both sharing the data among research teams and at the same time preserving the anonymity of the developers. The method used to achieve both goals simultaneously is consequence of the database design. This design implements three regions, each having a different distribution policy: personal information is private and is marked as for non distribution; information that gives links among the various identifications of a person is distributable under certain circumstances, mainly to other research groups, and contains no private data, only a way of knowing that two different identifications correspond to the same person; finally, a complete impersonal set of data is obtained that may be published publicly.

6.2.2 Reproduction and generalization of classical studies: Software Evolution

We have also evidenced that the availability of data from software projects can be used to reproduce *classical* software engineering analyses that rely on the availability of source code. This is the case for software evolution, especially in regard to the software growth analysis of software systems once they have been delivered for their first time. For the last thirty years, researchers have studied the growth pattern of (closed) industrial software systems yielding some results that have taken the form of several *laws* as they apply in general to all software programs. Thus, our first efforts have been to reproduce the software evolution studies on libre software. Nevertheless, we have not been the first ones to do a replication study on libre software as other research studies had already done that before. What is a contribution of this thesis in this sense is the generalization of these studies to many libre software projects. Up to now, the largest number of libre software projects that had been researched for their growth patterns at a time was three. In this thesis we have studied 21 large libre software projects, so the sample has been increased considerably. The results obtained for these projects give that linear patterns are the most frequent one in contradiction with some of the *laws* of software evolution as known so far. Although the number of selected libre software projects is large enough, other projects should be considered in future research to support this finding.

Besides replication, in this thesis we have widen the range of files considered in evolutionary studies. This opens the door to a richer set of analyses. So, we have generalized the concept of evolution to source artefacts different than those containing source code (i.e. files that contain code written in programming language by humans), including elements in the analysis such as documentation, translations, user interface files, among others. These type of files exhibit specific, sometimes peculiar, behaviors that differ from the one known for source code files. In our opinion, they should be properly attended as they are also part of the sources of a software to be maintained by the development team. In this regard, we have seen that some of the concepts in use for code can be applied to these other elements, as it is the case for (too much) common coupling, detected in files that are frequently changed together. A further step could be to identification coupling within files of different types.

Another innovative contribution of this thesis has been to adapt the concept of software evolution to the world of software compilations. Compilations are in charge of integrating vast amounts of software into a unique, homogene body. Our analysis has been performed by means of a case study on the Debian Linux-based distribution, which has been researched for its current state and how it has been evolving in the last seven years. We have seen how a software compilation evolves in number of packages (software programs) or in number of developers. Other information can be obtained from the importance of the various programming languages over time and the composition of the largest packages. In addition, derivative metrics that give a numerical idea of the work load that contributors suffer in the Debian project have been proposed and applied. The results of these metrics have been discussed, especially considering if they should be considered an assumable challenge in the near future for the Debian project.

6.2.3 A new way of extracting data from software versioning repositories: Software Archaeology

In this thesis, we have introduced the concept of *software archaeology* which provides a new perspective on the software development and maintenance process that is, as discussed in section 4.4, complementary to the one given by software evolution. The main idea of software archaeology can be summarized as follows: by means of measuring how many lines remain from the past in the current state of the software, we can infer the maintenance effort that a software has undergone in recent times. At the same time, we have briefly discussed how we this information could provide us means of estimating the future maintainability of the project.

We have applied to several libre software projects a methodology that allows to infer archaeological information from projects using versioning systems from a macroscopic point of view. The results have been shown in a tabular and a visual form by means of plots, but the creation of additional indexes has been introduced to manage the high amount of information that is available. Nevertheless, it should be noted that further research should attend the ideas and the results that can be inferred from

software archaeology. This thesis does not deepen in it, although it shows that software archaeology is a promising path to go.

A line that we have briefly commented for the possible uses of information gained by a software archaeology analysis is the knowledge that it provides about developers involved in a project. The key idea behind this issue is that the maintenance of a software should be easier if the original author is part of the maintenance team. Hence, we propose a way of quantifying the *importance* of a developer in a project and how much knowledge this developer has of the project. If we sum up the individual measures for developers in a project, we could use it, for instance, to infer how much *accumulated knowledge* the current team has.

6.2.4 Analysing technical artefacts from the social perspective

In this thesis we have widely studied socio-technical and social parameters related to the development of software, especially in attention to libre software projects. In these environments we have observed patterns of flexible management and self-organization. So, for instance, the participation of volunteers in tasks that are usually self-selecting conform also a challenge for the understanding of the processes that take place.

From the technical point of view, our interest in the social structure of software projects lies in the assumption that (technical) innovation in general follows the (or a similar) structure as the organization that performs it (this is also known as Conway's *law*, as presented in subsection 2.4.1). Hence, a software architecture is consequence of how a project is organized. We have therefore introduced some common techniques from social network analysis, yielding some interesting results. In this sense, we have found that -as suspected- large libre software projects fulfill the requirements to be considered small worlds, indicative for a structure with good information flow. The high amount of interconnections has made us introduce weights for the relationships in the developer and modules networks that we have built. Based on the weights we have used recent algorithms that allow to infer the structure of a network and have applied it to some software projects over time, obtaining a good perspective of how the project structure is in its current state and how it has evolved from its beginnings.

But other key concepts have been handled: territoriality, specialization and communities. Territoriality has been introduced to see how many developers collaborate on files for each type, trying to find out if files are *touched* by one or two developers (indicative for high territoriality) or by more developers. Gate-keeper effects and distribution of work can be identified by means of this analysis. Further insight is provided by the study of specialization, concerned with the number of file types that developers work on; if, in general, they attend many file types then the specialization in a project is low, while if they focus on a reduced set of file types we have specialized developers. Companies wanting to invest on a libre software development project could use this information to see where their investment could be more beneficial; if low specialization is the case, specialized contributions could be required.

A longitudinal analysis, as the one we have performed in this thesis, has given us evidence about project members becoming more specialized over the years. We have also studied the various commiter communities that arise in the project, with members of each community targeting specific tasks. By means of scatter plots we have showed a graphical way that allows to infer how these communities relate one to each other.

Continuing with longitudinal analysis, we have focused on the abandonment of libre software projects by developers. We have studied the impact of having contributions done mostly by volunteers using a well-known libre software distribution, the Debian Linux-based distribution, as a case study. We have observed that, at least for our case study, where contributors have to pass an admission procedure to form part of the project, the time span in which they contribute is long. We have therefore introduced the concept of developer half-life, a measure that gives the time in which a given community falls to half of its initial size.

On the other hand, we have introduced a methodology that allows to see if there are different generations that take the lead in a project. This can be seen as the complementary view of the half-life concept, which measured the fall in number of the community. In this case, what we are looking

for is the regeneration of the project, with various sets of developers over time who take the lead. In this regard, we have also shown that by means of analyzing the data sources, we can gain some insight into the integration process. We have tracked developers of a large software project used as case study and have seen how their activity patterns are from their first contributions until they became major contributors to the project.

6.2.5 Modelling libre software development using an analogy from the biological world

Although the main scope of this thesis is more related to empirical research, a model that tries to explain the development of libre software *in the large* has been proposed. In comparison to other suggested models, the innovative contribution of our model is that it is based on an analogy from the biological world. Therefore, our inspiration can be located from biology research in the field of social insects, mainly ants, termites and some sorts of spiders. Even social insects organize themselves by means of simple social structures, they achieve to build complex structures. Biologists have found that this is due to the use of distributed stimuli and not through a hierarchical structure.

As we considered that these characteristics can also be found in the development of libre software, we have adapted a well-known model used for ants looking for food to the libre software environment. In it, developers participate in projects. Decisions are taken stochastically depending on a set of factors that depend on the environment (and that act as *stimuli* for the developer). The quantitative analysis that is the main part of this thesis has been used to verify and discuss the accuracy of our model, yielding in general good results.

6.3 Limitations

Although throughout this thesis we have seen that the amount of data at our disposal is large and that we can apply to them an unlimited number of methodologies to extract knowledge about the projects under study, our approach has certain limitations which should be considered.

The main limitations is that there exists information that cannot be obtained from publicly available sites. In other words, we will never have a complete information space. For instance, even in the case of having a data environment as rich as the ones that we have found for some libre software projects, we are not capable of acquiring information such as the motivation that drives developers to participate in a project (or why he participates in a given project and not in another one), if the participation of the developer has been due to an economic transaction, and many other philosophical and political questions that might be interesting for the complete understanding of the phenomenon.

It is true that surveys can provide here some information that might help fill the gap, as we have shown in the chapter devoted to the related research where some of the already performed questionnaires have been presented (see section 2.4.3). But surveys are limited in scope and in time as they are intrusive and will always be limited in number of questions and in the number of respondents that answer them.

On the other hand, there exists information that is beyond the one that can be obtained through the analysis of public data sources or by means of surveys. This information is, for instance, the time they devote to a certain task; we have the possibility of knowing the actions that have taken place by analyzing logs and we could probably obtain an approximation of the time devoted by developers.

Another limitation is the one due to an incomplete or partial data set. This may be the case for old projects where archives have been lost or removed or when development-supporting tools currently in use were not used. This is a handicap that all empirical studies based on historical data sets have to confront and thus not new to software engineering research.

A final limitation is preservation of the privacy of those persons who are active in software projects. We have seen in this thesis a proposal for integrating data about developers from various data sources while at the same time preserving (at least partially) the privacy of the involved developers. But in the future, the public access to the data sources together with the improvement of the tools will probably make any attempt to preserve privacy very difficult. Beyond any doubt, this is a question that

goes beyond software engineering practices and will also affect search engines and other information manipulating resources on the Internet.

6.4 Further research

This thesis has been a first approach to the data sources that are publicly available. We have identified and described some data sources and, based on the data they provide, have shown some possible methodologies and analyses. Without doubt, this is only a first step towards using these data for research purposes. In this sense, some of the following ideas (grouped as further research) summarize some interesting research lines that our analysis have thrown.

Many (if not all) of the methodologies presented in this thesis have been supported with some case studies. As we have the methodology and the tools available, the next step should be to generalize results by studying large amounts of projects (from the libre software realms, but also, if possible, from industrial environments). A large-scale investigation could have as objective to obtain an ontology that would help characterize software projects and group them into *families*. This would have several promising side effects: the first ones is that having software categories will help to (re)use knowledge and experience from similar projects.

Further research should also put its attention to the relationship between the proposed idea of software archaeology and software evolution. We have seen that software archaeology provides an alternative point of view of a project than the one offered by software evolution, but in this thesis we have not look in profundity into how both views are related.

This thesis has also shown that there is a wide range of other source artifacts that deserve to be researched as they may help understanding the development process. In this sense, we have seen in the related research that source code and versioning systems are by far the most studied domains. New research lines should embrace also documentation, translation, user interface elements, among others, and include data sources as mailing lists, bug-tracking systems and others.

In this direction, the next step should be to integrate information from various sources. We have seen several approaches in literature so far and this thesis has proposed another one. But, again, we are the beginning of what can be done. We have seen that the possibilities that arise from integrating data from different sources are very interesting and provide new perspectives and evidences hardly obtainable from a single one or from considering sources in a separate way.

Regarding the knowledge on the social structure of projects, the findings related to specialization and communities that have been presented in this thesis are especially interesting for non-hierarchical environments like the ones found in the libre software world, where self-organization and volunteer work applies. Hence, we find that further research could help lowering the gap between the software industry and the libre software world and should focus on finding out how software companies could benefit from this type of environments with the help of analysis like the ones proposed here.

From a more technical point of view, an interesting follow-up could be to deepen in the discrimination of the files, attending to their content and widening the classification. So, code files could be sorted by programming language. New communities could be identified and new relationships would probably arise. Another possible research line is the identification of groups of files that are regularly part of the same atomic commits. This has already been studied in literature for source code, and we have seen some hints about this for other file types in this thesis. But a more general study could be made, attending not only to the study of coupling for each file type on its own, but also considering coupling among them.

One of the main drawbacks of this dissertation is that it could be considered as basic research. In this sense, application studies should emphasize in the power of this approach, helping in how industry can benefit from all these information and methodologies. The applicability in the short term could be to help industry asserting the quality of a libre software project. A question that occurs frequently is a company wanting to invest or to use a libre software applications, but that has to study the many applications that exist. Providing information about the software itself as well as a set of measures related to the community could be a great step forward. Community metrics could help to ascertain, for instance, the support that this community is ready to give.

On another hand, as we can track the activities that have been performed and we have shown in this thesis how to do it in several sources for a given developer, future research could be devoted to estimate the effort that has been put on a (libre) software project. This may serve as a good starting point for cost estimation in libre software projects. It should be noted that our methodological approach allows to track activity around the development of libre software. By means of a detailed analysis of the various activities that take place (development, maintenance (and its various types), support in mailing lists, among others) we may infer the effort that a software has supposed. Once the effort is known, and with the help of techniques presented in this thesis that permit to ascertain the geographical situation of a developer, we could infer the cost. A very promising path may arise from this research line.

6.5 Final summary

This thesis is a first step in the use of vast amount of publicly software development information. We have described the most common public data sources that are available for inspection on the Internet, have described methods to extract data from then and have proposed a set of methodologies (adapting methodologies from related research or proposing new ones) for the analysis of the data. This thesis has demonstrated that probably for the first time in the history of software engineering, researchers have the possibility of accessing large amounts of data sources that may help understanding the underlying development processes better, that may serve to benchmark new proposals and that widens many new possibilities regarding effort estimation and social interactions. Exhaustive analyses of many software projects, as it has been done in this thesis, with the help of tools and methods is a very promising path and will provide future software engineering research with higher accuracy and richer views.

Appendix A

Applications under consideration

This appendix introduces shortly the software projects that have been used as a case study throughout this thesis. Most of the projects, if not all, share the characteristic of being large in software size (above the 100 KSLOC) and in number of contributors (with hundreds of participants). Thus, these projects can be considered to have been developed using the *bazaar* development model.

- **Ant** is a Java-based build tool similar to traditional Make. It is developed by the Apache Software Foundation.
- **Apache 1.3** is the most-used HTTP server world-wide. Although its development is frozen as the active branch is the Apache 2.0 one, Apache 1.3 is still the most used Apache version nowadays[1]. Apache has been the target of several research studies [Mockus *et al.*, 2002; Paulson *et al.*, 2004].
- **Apache 2.0** (httpd 2.0) is the next generation of the most-used HTTP server world-wide, Apache. Apache has been the target of several research studies [Mockus *et al.*, 2002; Paulson *et al.*, 2004].
- **Emacs** is an editor written in LISP from the 1970s but still very popular, although it was completely rewritten by Richard Stallman in the mid 80s. We assume it can be treated as a libre software legacy system, as it has been in a mature state for around 20 years now.
- **Evolution** is a libre group-ware solution by Novell part of the GNOME project which contains e-mail application, calendar, etc. It is a case sample of a corporate-driven development that has achieved to group a community around it. It has been studied before in detail by German [Germán, 2004a; Germán & Mockus, 2003].
- **galeon** is a web client for GNOME based on Gecko, the Mozilla engine.
- **GCC** is the GNU Compiler Collection which contains compilers for C, C++, Java, ADA, etc. Many entities are interested in the development of GCC and it has had a complicated history with several forks and merges of compilers, so large amounts of third-party code have been imported to the repository. There have been previous studies on GCC [Paulson *et al.*, 2004].
- **Gnumeric** is a spreadsheet application for the GNOME project.
- **GTK+** is the Graphical ToolKit developed for The GIMP. It was taken over by the GNOME community and serves since then as the graphical toolkit of the whole GNOME desktop environment and not only of The GIMP. This makes it specially interesting, as there is a high number of projects which depend on it (in opposition to The GIMP). The old code base from which less than 18,500 lines remain today was imported into the current repository in December 1997.

[1] Any of the Apache 1.3 versions is used by at least 50% of all httpd servers worldwide and over 66% of all Apache servers installed. Source: Web Server Survey - Server Breakdown (published March 1st 2005): http://www.securityspace.com/s_survey/data/200502/servers.html

- **Jakarta Commons** is a repository for small scale, reusable, code components that are useful in multiple Jakarta subprojects. It is part of the Jakarta Project which offers a set of Java-based solutions. The Jakarta Project is part of the Apache Software Foundation.
- **kdebase** is in addition to kdelibs the second mandatory package for the use of the KDE desktop environment. It contains several applications, infrastructure files and libraries.
- **kdelibs** contains the core libraries for the KDE desktop environment. It has been selected as it is a project where a large quantity of developers have contributed to. Although it is not a graphical tool kit as GTK+ (KDE uses the Qt toolkit) it can also be seen as a project on which many others rely on.
- **kdenetwork** groups applications for networking for the KDE project.
- **kdepim** is the Personal Information Management suite for the KDE project.
- **KDevelop** is an Integrated Development Environment (IDE) for the KDE project.
- **KOffice** is an office suite for the KDE environment. It groups a set of applications as word processor (KWord), spreadsheet (K), etc.
- **MCS** contains a hierarchy of classes for the C# language for the MONO Project, a libre implementation of the .NET Developer Framework.
- **Mono** is the the C# compiler for the MONO project.
- **Mozilla** is a well-known Internet suite which groups web navigator, e-mail client, etc. It is the libre software successor of the Netscape Navigator suite and is the canonical example for an application that has been released to the community once an ample code base already existed. This happened in April 1998; almost 154,000 lines of code remain from then. It is also one of the most studied libre software projects [Antoniol *et al.*, 2005; Ferenc *et al.*, 2004; Fischer *et al.*, 2003; Mockus *et al.*, 2002].
- **The GIMP** comes from GNU Image Manipulation Program and is the most-known libre competitor of Photoshop. It was started in the mid-90s by two students at Stanford who left the project after releasing the 1.0 version, but has been taken over by other developers and is today a community-driven project. The initial import into the current CVS happened December 1997 with around 39,000 lines remaining from then.
- **WINE** is the recursive acronym for WINE Is Not an Emulator, although we could say it is a Windows emulator for UNIX systems. Its most interesting characteristic is that its development is not community-driven, so a limited amount of developers from the company who builds WINE have directly touched the repository. Wine is the unique application considered which has not achieve a 1.0 version.
- **XML:Xalan** is an XSLT processor for transforming XML documents into HTML, text, or other XML document types. It is a project that is promoted by the Apache Software Foundation.
- **Xerces-C++** is a validating XML parser written in a portable subset of C++. Its main goal is to make the process of reading and writing XML data easy. Xerces-C++ is part of the Apache project.

The Debian distribution

Debian is a libre operating system that, at present time, uses the Linux kernel to carry out its distribution (although there are some efforts to make that future Debian distributions could be based on other kernels, such as The HURD or even FreeBSD). At the moment Debian is available for several architectures, including Intel x86, ARM, Motorola, 680x0, PowerPC, Alpha and SPARC.

Debian is not only the largest GNU/Linux distribution at present time, it is also one of the most stable and enjoys several prizes based on user preferences. Although its number of users is difficult to estimate, since the Debian project does not directly sell CDs and the software that it contains can be redistributed by anyone who desires to do so, we can assume, without any doubt, that it is an important distribution within the GNU/Linux market.

Debian has a categorization of software packages according to their license and their distribution requirements. The main part of the Debian distribution (the section called *main*, which contains a large variety of packages) is compound only of libre software in agreement with the [DFSG] (the Debian Free Software Guidelines). It is available for download from the Internet and many resellers supply it on CDs or by other means.

The Debian distribution has been created by around a thousand volunteers (generally computer professionals). The work of these volunteers consists on taking the source programs -in most of the cases from their original author(s)-, to configure them, to compile them and to pack them, so that a typical user of a Debian distribution only has to select the package to be installed/updated/removed. In principle, the volunteers' work may seem an easy task but, as soon as factors like dependencies between packages (the package A needs, to be able to work, of package B) or the existence of different versions for all these packages are introduced, the process shows higher complexity.

The work of the members of the Debian project is similar to the one carried out in any other distribution: software integration. In addition to adaptation and packing work, the Debian developers are in charge of the maintenance of services based on Internet infrastructure (web site, on-line archives, bug management systems, mailing lists, support and development, etc.), for several translation and internationalization projects, for the development of several Debian-specific tools and, in general, for any element that makes the Debian distribution possible.

Besides its voluntary nature, the Debian project has a characteristic that makes it specially singular: the Debian Social Contract [DebianSocialContract]. This document contains, not only the primary goals of the Debian project, but also the means that will be used to carry them out.

Debian is also well-known to have a very strict package and versioning policy with the purpose of obtaining a better product quality [DebianPol]. Thus, at any moment three different *flavors* of Debian exist: stable, unstable and testing versions (it also exists another *version*: Debian experimental, which will be treated next). As its name points out, the stable version is targeted to systems and people looking for high stability. Its software has to pass a freezing period in which only critical errors are corrected. The rule is that when a stable version is released it should not contain any known critical error. On the other hand, because of the freezing period, stable versions usually do not include the most recent software versions.

For the ones that wish to have a version with the current software other two contemporary versions to the stable one exist. The testing version includes packages that are on the way of becoming stabilized, whereas the unstable version, as its own name points out, is more inclined to fail and contains the latest of the latest on libre software applications and tools.

Recently, another *flavor* reached to Debian world: it is named as 'experimental' and it includes packages in early stages of development. So, these packages must be considered as really 'unstable'; while 'unstable' branch can be used by most advanced users.

The code-names of the versions in Debian correspond to the main characters of the animated cartoon film *Toy Story*, a tradition that started with version 1.1 when Bruce Perens, then leader of the Debian project and later founder of the Open Source Initiative and the term Open Source, worked for the company that was in charge of producing these movies. More details on the history of Debian and the Debian distribution in general can be found in [DebianHistory].

Appendix B

File extensions

This appendix contains a list of the file extensions and filenames that have been used in the discrimation process described in theory in subsection 3.2.2 and applied empirically to KDE as a case study in section 4.5. The use of the ? and * symbols follows the usual pattern matching conventions.

Development file extensions	
.c	C source code file
.pc	C source code file
.ec	C source code file
.ecp	C source code file
.C	C++ source code file
.cpp	C++ source code file
.c++	C++ source code file
.cxx	C++ source code file
.cc	C++ source code file
.pcc	C++ source code file
.cpy	C++ source code file
.h	C or C++ header file
.hh	C++ header file
.hpp	C++ header file
.hxx	C++ header file
.sh	Shell source code file
.pl	Perl source code file
.pm	Perl source code file
.pod	Perl source code file
.perl	Perl source code file
.cgi	CGI source code file
.php	PHP source code file
.php3	PHP source code file
.php4	PHP source code file
.inc	PHP source code file
.py	Python source code file
.java	Java source code file
.class	Java Class source code file
.ada	ADA source code file
.ads	ADA source code file
.adb	ADA source code file

.pad	ADA source code file
.s	Assembly source code file
.S	Assembly source code file
.asm	Assembly source code file
.awk	awk source code file
.cs	C# source code file
.csh	CShell (including tcsh) source code file
.cob	COBOL source code file
.cbl	COBOL source code file
.COB	COBOL source code file
.CBL	COBOL source code file
.exp	Expect source code file
.l	(F)lex source code file
.ll	(F)lex source code file
.lex	(F)lex source code file
.f	Fortran source code file
.f77	Fortran source code file
.F	Fortran source code file
.hs	Haskell source code file
.el	LISP (including Scheme) source code file
.scm	LISP (including Scheme) source code file
.lsp	LISP (including Scheme) source code file
.jl	LISP (including Scheme) source code file
.ml	ML source code file
.ml3	ML source code file
.m3	Modula3 source code file
.i3	Modula3 source code file
.m	Objective-C source code file
.p	Pascal source code file
.pas	Pascal source code file
.rb	Ruby source code file
.sed	sed source code file
.tcl	TCL source code file
.tk	TCL source code file
.itk	TCL source code file
.y	Yacc source code file
.yy	Yacc source code file
.idl	CORBA IDL source code file
.gnorba	GNOME CORBA IDL source code file
.oafinfo	GNOME OAF source code file
.mcopclass	MCOP IDL compiler class source code file
.autoforms	Autoform source code file
.atf	Autoform source code file
.gnuplot	GNUplot file
.xs	Shared library (gnome-perl)
.js	JavaScript source code file
.patch	Patches
.diff	Patches
.ids	CORBA source code file
.upd	(from Kcontrol)
.ad	from Kdisplay and mc
.i	Appears in the kbindings for Qt
.pri	from Qt

.schema	
.fd	LaTeX
.cls	LaTeX
.pro	Postscript generation
.ppd	PDF generation
.dlg	ArcView Digital Line Graph File
.plugin	Plug-in file
.dsp	MS Developer Studio Project
.vim	vim syntax file
.trm	gnuplot term file
.font	Font mapping
.ccg	C++ files (gtkmm*)
.hg	C++ headers (gtkmm*)
.dtd	XML DTD
.bat	DOS batch files
Development documentation file extensions	
readme.*	README information file
changelog.*	Change log file with the changes applied
todo.*	File with a list of activities to do
credits.*	File with authorship information
authors.*	File with authorship information
changes.*	Similar to ChangeLog files
news.*	Similar to ChangeLog files
install.*	Installation instructions
hacking.*	File with coding conventions or activities to be done
copyright.*	Copyright or license files
licen(s—c)e.*	File with the license text
copying.*	File with the license text
manifest	File with general information about the software
faq	File with the frequently asked questions
building	Building instructions
howto	
design	
.files	
files	
subdirs	
maintainers	Authorship information
developers	Authorship information
contributors	Authorship information
thanks	Authorship information
test	
testing	
build	
comments?	
bugs	File with a list of known bugs
buglist	Files with a list of known bugs
problems	Files with a list of known bugs
debug	Files with a list of known bugs
hacks	File with coding conventions or activities to be done
hacking	File with coding conventions or activities to be done
versions?	
mappings	

tips	
ideas?	
spec	
compiling	Compiling and building instructions
notes	
missing	
done	ChangeLog file
.omf	XML-based format (GNOME)
.lsm	Linux Software Map
.kdevprj	KDevelop project
.directory	
.dox	MultiMate Document
Building, compiling, configuration and CVS administration file extensions	
.in.*	Build-process configuration instructions
configure.*	Build-process configuration instructions
makefile.*	Makefile instructions
config.sub	Build-process configuration instructions
config	Build-process configuration instructions
conf	Build-process configuration instructions
cvsignore	Contains a list of files to be ignored by CVS
.cfg	Configuration file
.m4	ASCII M4 macro language pre-processor text
.mk	Makefile
.mak	Makefile
.make	Makefile
.mbx	e-mail file type associated with Outlook and Eudora
.protocol	
.version	
mkinstalldirs	
install-sh	
rules	
.kdelnk	KDE Desktop Link
.menu	
.shlibs	Shared libraries
.spec	Necessary for RPM building
Documentation, web-pages file extensions	
.html	HTML page
.txt	Text File
.ps	PostScript file
.dvi	Action Media II Digital Video Interactive Movie
.lyx	WYSIWYM document processor
.tex	Tex filetype
.texi	Tex filetype
.pdf	Acrobat Portable Document Format
.sgml	Standard Generalized Markup Language IETF Document
.docbook	DocBook file
.wml	Website META Language File
.xhtml	Extensible HyperText Markup Language File

.phtml	Embedded Perl (ePerl) File
.shtml	HTML File Containing Server Side Directives
.htm	Hypertext Markup Language
.rdf	Burli Newsroom System File
.phtm	PHP Script
.tmpl	formVista Template
.ref	References
.css	Hypertext Cascading Style Sheet
.templates	
.dsl	DSSSL Style Sheet
.ent	Brookhaven PDB Molecule File
.xml	Extensible Markup Language File
.xsl	FileMaker Database Design Report Template
.entities	
.man	Manual help page
.manpages	Manual help page
.doc	Word Document
.rtf	Rich Text Format File
.wpd	602 pro PC SUITE Document File
.qt3	
man?.*.	Manual help page
.docs	
.sdw	OpenOffice.org Writer document
.en	Files in English language
.de	Files in German
.es	Files in Spanish
.fr	Files in French
.it	Files in Italian
.cz	Files in Czech
Image and graphics file extensions	
.png	Portable (Public) Network Graphic
.jpg	JPEG/JIFF Image
.jpeg	JPEG/JIFF Image
.bmp	Windows OS/2 Bitmap Graphics
.gif	Graphic Interchange Format
.xbm	X Bitmap Graphic
.eps	Encapsulated PostScript
.mng	Multi-image Network Graphic Animation
.pnm	PBM Portable Any Map Graphic Bitmap
.pbm	UNIX Portable Bitmap Graphic
.ppm	PBM Portable Pixelmap Graphic
.pgm	Portable Graymap Graphic
.gbr	Gerber Format File
.svg	Scalable Vector Graphics File
.fig	Geometry II Plus Figure
.tif	Tagged Image Format File
.swf	Macromedia Flash Format File
.svgz	Compressed Scalable Vector Graphics File
.shape	XML files (Kivio)
.sml	XML files (Kivio)
.bdf	vfontcap Backup To CD-RW Backup Definition File
.ico	Windows Icon

i18n and l12n file extensions	
.po	GNU Gettext Portable Object
.pot	
.charset	
.mo	
User interface file extensions	
.desktop	
.ui	Geoworks UI Compiler Espire Source
.xpm	BMC Software Patrol UNIX Icon File
.xcf	GIMP Image File
.3ds	3D Studio 3D Scene (Mesh File)
.theme	Theme File
.kimap	
.glade	GNOME UI Builder
rc	
Multimedia file extensions	
.mp3	MPEG Audio Stream, Layer III
.ogg	Ogg Vorbis Codec Compressed WAV File
.wav	Waveform Audio
.au	uLaw/AU Audio File
.mid	Musical Instrument Digital Interface MIDI-sequention Sound
.vorbis	Audio format
.midi	Musical Instrument Digital Interface MIDI-sequention Sound
.arts	

Appendix C

Resumen en español

No he visto todavía ningún problema, tan complicado como sea, que cuando lo miras de la manera correcta, no se vuelva todavía más complicado.
Paul Anderson

Esta tesis trata sobre el estudio de información de desarrollo pública, en particular de proyectos de software libre, desde un punto de vista de ingeniería de software. Esto es interesante por (al menos) dos motivos. Por un lado, el software libre provee -probablemente por primera vez en la historia de la ingeniería del software- una gran cantidad de información pública. En estos entornos de desarrollo por lo general no sólo el código fuente está disponible, sino que también muchos de los subproductos del desarrollo como, por ejemplo, el contenido de los principales canales de comunicación (listas de correo, foros, etc.) o de las herramientas de soporte al desarrollo (gestores de erratas, sistemas de versiones, etc.). Por otro, en los últimos años hemos sido testigos del éxito de al menos un número no despreciable de proyectos de software libre que han sido desarrollados siguiendo unos principios de desarrollo poco comunes en la industria del software. El conjunto de prácticas suele incluir (de manera conjunta): acceso al código fuente en desarrollo, desarrollo global[1], revisión por pares, prototipado rápido, auto-organización, estrategias para facilitar contribuciones externas, etc.

Esta tesis tiene como objetivo mostrar análisis y resultados a partir de la información pública (la primera razón) en aras de comprender el proceso de desarrollo que se da en el software libre (la segunda razón). Debido a la complejidad intrínseca de los procesos asociados al desarrollo software, veremos que hará falta un análisis polifacético.

C.1 Antecedentes

El software libre ha recibido, sin lugar a dudas, mucha atención últimamente por parte del mundo académico, de la industria del software, de gobiernos y administraciones públicas y de usuarios finales. Aunque sus principios filosóficos datan de los años ochenta[2], no ha sido hasta hace unos pocos años cuando con la proliferación de conexiones a Internet se ha mostrado como un modelo de desarrollo y de distribución de software exitoso. Existe un amplio debate sobre las implicaciones y futuras posibilidades que ofrece el software libre, hasta el punto de que algunas voces indican que se está convirtiendo en una *amenaza* competitiva para los modelos de desarrollo -cerrados y pesados- y de distribución -orientada al pago de licencias- del software tradicional.

Y es que el conjunto de métodos y prácticas seguidos por algunos proyectos de software libre ha despertado recientemente interés, primero entre la propia comunidad del software libre y posteriormente entre equipos de investigación del campo de la ingeniería del software, entre otros. De los primeros trabajos descriptivos y en parte ensayísticos sobre la naturaleza del desarrollo del software libre hemos pasado en un breve espacio de tiempo a estudios empíricos basados principalmente en datos obtenidos a partir de proyectos de software libre o en casos de estudio de los proyectos. Es interesante

[1] Global Software Development (GSD) es el término original en inglés.

[2] Habría que hacer notar, sin embargo, que el software libre existía *de facto* con anterioridad, estando íntimamente ligado a los primeros pasos del propio software [Stallman, 1999; Salus, 2005].

indicar, sin embargo, que el software libre es en definitiva una condición legal plasmada en la licencia con la que se distribuye el software. No hay, por tanto, una forma de desarrollo específica, sino que cada proyecto ha ido creando su propia manera de organizarse y de tomar decisiones. En consecuencia, no existe un modelo único de desarrollo de software libre, aunque bien es cierto que muchos de los proyectos que han tenido éxito cuentan con una serie de características en común o al menos similares, generalmente debido a que la apertura del proceso de desarrollo plantea muchas ventajas competitivas.

Un concepto muy popular, introducido por Raymond en uno de los primeros estudios/ensayos sobre desarrollo de software libre, es el de *bazar* [Raymond, 1998]. La idea detrás del *bazar* consiste en tener un modelo de desarrollo que funcionara de manera similar a como lo hacen los bazares orientales. En ellos, los intercambios ocurren de manera espontánea y no están dirigidos por nadie. Asimismo, los roles se intercambian frecuentemente y no hay planificación centralizada. En contraposición a cómo funcionan los bazares, Raymond argumenta que las prácticas de desarrollo software contemporáneo se asemejaban a la manera en la que se construían *catedrales* en el medievo. En el modelo de *catedral* tenemos roles y tareas perfectamente especificadas y existe una gestión planificada y central. Sin embargo, esta forma de crear software no es exclusiva de empresas de software que crean software propietario, sino que Raymond indica que algunos proyectos de software libre también seguían este tipo de prácticas - lo que redunda en la puntualización hecha con anterioridad, en la que se apuntaba que la licencia con la que se distribuye el software no implica necesariamente que se haya desarrollado de una manera u otra.

Aún así, muchos de los primeros esfuerzos consistieron en comparar el desarrollo tradicional (en cascada o V [Royce, 1970]) con el *bazar*, que se suponía era el modelo de desarrollo más indicado para proyectos de software libre. Se argumentaba que la falta de formalidad en el proceso de desarrollo era compensada por la (rápida y abundante) realimentación de los usuarios y la posibilidad de integrarlos en el ciclo de desarrollo, así como de la adopción de manera dinámica de métodos y prácticas de desarrollo según vaya evolucionando el proyecto [Vixie, 1999].

En cualquier caso, los primeros estudios realizados fueron generalmente descriptivos y aunque ciertamente han influenciado de manera notable para bien y para mal el desarrollo de la investigación del software libre, casi todos ellos se basaban en percepciones personales y adolecían de datos objetivos.

Ha sido a principios de la década actual cuando la comunidad investigadora se ha dado cuenta de la gran cantidad de datos disponibles para ser analizados. Uno de los primeros estudios en este sentido se realizó con el objetivo de conocer la distribución de la autoría del software en los proyectos [Ghosh & Prakash, 2000]; los resultados mostraron que a un pequeño grupo de personas le correspondía la autoría de una gran cantidad del código fuente. Este comportamiento ha sido descrito también para otro tipo de fuentes de datos -principalmente mensajes a listas de correo e interacciones con el repositorio de versiones- por parte de Koch et al. [Koch & Schneider, 2002] en un estudio cuya finalidad era comprobar la validez de los modelos de predicción de costes en el software libre.

Pronto se vio que había que distinguir claramente entre proyectos grandes de software libre - aquéllos que conseguían tener una masa crítica de desarrolladores y usuarios a su alrededor- y el resto de proyectos, más modestos en número de participantes. Krishnamurthy estudió cien proyectos *maduros* hospedados en SourceForge, uno de los sitios web que integran herramientas de desarrollo para proyectos de software libre y cuyos servicios se pueden utilizar gratuitamente, en el que demuestra que la gran mayoría de los proyectos contaban con un pequeño número de desarrolladores [Krishnamurthy, 2002]. Existen, por tanto, dos tipos de estudios sobre proyectos de software libre: los que atienden a todos los proyectos de software libre (como por ejemplo [Healy & Schussman, 2003; Howison & Crowston, 2004; Conklin *et al.*, 2005]) y los que se han centrado en proyectos de software libre grandes (en líneas de código y participantes), ya que éstos muestran las características más interesantes tanto desde el punto de vista ingenieril como socio-económico.

Mockus et al. [Mockus *et al.*, 2002] realizaron un profundo estudio de la composición de la comunidad en un proyecto grande y comprobaron cómo los proyectos de software libre constan de un pequeño grupo de desarrollo que se encarga de la mayoría del trabajo, de las tareas técnicamente más importantes y de la gestión y organización del proyecto. Este grupo - conocido como el núcleo o *core*- está formado por unas doce o quince personas. En torno al núcleo existe una serie de colaboradores habituales cuyo número es un orden de magnitud más grande. Finalmente, existe otro grupo de

colaboradores ocasionales que es a su vez en número un orden de magnitud más grande que el de habituales. Esta estructura ha recibido el nombre del modelo de cebolla [Crowston *et al.*, 2003a], ya que se suele representar con el núcleo en el centro rodeado concéntricamente por el resto de colaboradores, cuanto más cerca del núcleo más participativos. El modelo de cebolla es un modelo estático, ya que no tiene en cuenta cambios en el tiempo en la composición, por lo que algunos autores le han dado dinamicidad para incluir la integración de nuevos miembros [Ye *et al.*, 2004], así como los procesos y los consiguientes cambios de roles que existen en proyectos de software libre en el tiempo [Scacchi, 2004; Jensen & Scacchi, 2005].

Este tipo de estudios que usan bases de datos con datos históricos sobre el desarrollo de un sistema software, convergen con iniciativas llevadas a cabo a finales de la década de los noventa en entornos corporativos [Gall *et al.*, 1997; Graves & Mockus, 1998; Atkins *et al.*, 1999; Mockus & Votta, 2000; Atkins *et al.*, 2002]. Por supuesto, en esos entornos la accesibilidad y la disponibilidad de datos no es tan fácil como en el caso de proyectos de software libre, por lo que muchos de los grupos de investigación que estaban investigando en el campo de la minería de repositorios software se han *mudado* en los últimos años al análisis repositorios de proyectos de software libre.

Por eso, el abanico de análisis de proyectos de software libre se ha abierto mucho durante los últimos años. Algunos esfuerzos han ido encaminados a reproducir estudios "clásicos" de ingeniería de software en entornos de desarrollo de software libre. Así, por ejemplo, dentro del campo de la evolución del software, uno de los casos más conocidos es el estudio del crecimiento del tamaño de Linux en el tiempo [Godfrey & Tu, 2000]. Este estudio demostró que Linux crece de manera superlineal, en contra (al menos) de una de las ocho *leyes* de evolución de software promulgadas por Lehman [Lehman & Belady, 1985; Lehman *et al.*, 1997].

Otros estudios han tomado herramientas y métodos tradicionales y los han aplicado al software libre. Un ejemplo paradigmático es un trabajo sobre la mantenibilidad de proyectos de software libre [Samoladas *et al.*, 2004], que utiliza los índices de mantenibilidad propuestos por Oman et al. a principios de los años 90 [Oman & Hagemeister, 1992; 1994] o trabajos dedicados al estudio en el tiempo del acoplamiento que existe entre las funciones de Linux y que hacen peligrar su mantenibilidad en el futuro [Schach *et al.*, 2002]. Otra muestra la da el uso de métodos del campo del análisis de redes sociales. Algunos autores han estudiado los vínculos entre los miembros de proyectos de software libre con el objetivo de determinar ciertas características de su comunidad de desarrollo. De esta forma, se ha podido estudiar la difusión de información intraproyecto o se ha procedido con la identificación de miembros de la comunidad que ostentan posiciones privilegiadas en su entramado social [Madey *et al.*, 2002; 2004].

También ha habido un amplio debate sobre la conveniencia del software libre en comparación con el propietario. El trabajo más conocido, aunque no el único, es el de Paulson et al., para el cual tomaron tres proyectos de software libre y tres proyectos de software propietario y los compararon según una serie de características del desarrollo (creatividad, complejidad, tiempo de corrección de errores y modularidad) [Paulson *et al.*, 2004]. Aunque el número de aplicaciones investigado es insuficiente para sacar conclusiones generales, en los casos estudiados los programas de software libre eran más creativos en comparación con los proyectos propietarios, pero más complejos, tardaban menos tiempo en corregir errores, pero eran menos modulares. El estudio combinaba el análisis estático de código fuente con elementos externos como el sistema de gestión de erratas y el sistema de control de versiones. Es importante destacar que muchas de estas relaciones se pueden obtener a partir de trazas del desarrollo del proyecto. Buen ejemplo de ello es un estudio realizado por German centrado en la información que se puede extraer al analizar las interacciones de los desarrolladores con un repositorio de versiones [Germán, 2004e; 2004a]. Esta visión muestra la existencia de unos periodos de tiempo dedicados al desarrollo y otros a la *limpieza* de erratas, el impacto de la modularidad del proyecto en la manera de trabajar e incluso cierta "territorialidad" del código, de manera que existen desarrolladores que tocan exclusivamente unos ficheros.

La posibilidad de redistribuir el software ha propiciado que se creen compilaciones de software libre, comúnmente conocidas como distribuciones, cuya finalidad es hacer la instalación y el mantenimiento de un sistema software fácil e intuitiva para el usuario final. Las distribuciones agrupan un gran

número de programas, de diverso tamaño y en diferentes lenguajes de programación. El estudio de distribuciones como Red Hat [Wheeler, 2000; 2001] o Debian [González-Barahona *et al.*, 2001; 2004; Amor *et al.*, 2005b; 2005a] se puede entender como una radiografía del software libre que se está utilizando mayormente, ya que éstas suelen incluir el software más demandado por los usuarios.

Aunque es cierto que podemos conseguir una cantidad de datos destacable de proyectos de software libre, sólo con ellos no estamos en condiciones de obtener una visión completa de este fenómeno. Es por eso por lo que otro tipo de estudios, complementarios a los empíricos, han de realizarse. Un buen ejemplo es un estudio, que los antropólogos calificarían de inmersión, en el que se analizó desde dentro las "costumbres" y las "reglas" de un proyecto de software libre de gran tamaño [Germán, 2004b]. Muchos de las consideraciones que ofrece este estudio permiten dar respuesta a interrogantes que aparecen a la hora de analizar resultados empíricos.

Asimismo, otra fuente de datos que permite completar el campo son encuestas realizadas a desarrolladores de software libre [Robles *et al.*, 2001; Ghosh *et al.*, 2002a; David *et al.*, 2003]. Gracias a las encuestas es posible conocer aspectos que son difíciles (sino imposibles) de obtener a partir de las fuentes de datos consideradas en esta tesis como pudieran ser las motivaciones que tienen los desarrolladores [Hertel *et al.*, 2003]. Por otro lado, también es posible completar información; por ejemplo, con los datos que generalmente tenemos, valorar el tiempo que los desarrolladores dedican a un proyecto de software libre es complicado, ya que contamos con acciones puntuales a partir de las cuales es difícil inferir el tiempo dedicado (envío de un correo electrónico o de un parche, etc.). Preguntar a los desarrolladores directamente por estas cuestiones nos permite añadir información de la que de otra forma no dispondríamos.

C.2 Objetivos

El objetivo de esta tesis es comprender los procesos y factores que tienen lugar durante el desarrollo de software libre. Para ello se empleará un punto de vista de ingeniería de software, con especial atención a métodos cuantitativos y empíricos.

Esta tesis debe enmarcarse dentro de los esfuerzos que se están realizando para conocer mejor el fenómeno del software libre. Debido a su reciente popularidad y a algunas de sus características (desarrollo distribuido, implicación de voluntarios, etc.), a la hora de abordar el estudio de software libre nos enfrentamos a una complejidad mayor que en el caso de desarrollos software tradicionales. Esto ha provocado que exista a la vez una cierta mitificación de este fenómeno, lo que unido a la ausencia de datos en los primeros estudios, ha hecho que muchas de las consideraciones emitidas con respecto al software libre se basen en opiniones y percepciones personales y no en datos y estudios contrastables y reproducibles. Por ello, se considera que una visión más ligada a los datos y a los hechos demostrables es crucial para entender mejor el fenómeno de software libre, algo que redundará en beneficio de la industria del software y de los propios participantes en proyectos de software libre.

En cierta medida, podemos encontrar en los exámenes médicos que se realizan a los pacientes una buena analogía con las ciencias de la salud. Nuestro objetivo es que, al igual que un doctor puede a partir de los datos de un análisis de sangre indicar el grado de salud de un paciente y qué valores deberían corregirse, podamos realizar análisis de proyectos de sobre libre. Nótese que para que esto sea posible, hacen falta que se cumplan dos condiciones: en primer lugar, que se puedan cuantificar factores relativos al desarrollo de un proyecto de software libre y, por otro, que podamos interpretar y clasificar los datos obtenidos, de manera que esté en nuestras manos identificar qué parámetros muestran tendencias poco *sanas* y que deberían ser corregidas. Por supuesto, la segunda condición requiere contar con el conocimiento necesario para saber que los valores que se dan son buenos o nocivos, así como cuáles son los métodos para cambiarlos en la dirección deseada. Esto sólo lo puede dar la experiencia a partir del análisis continuado de gran cantidad proyectos de software libre.

Esta tesis aborda principalmente la primera de las dos condiciones, la de la cuantificación de parámetros que caractericen a un proyecto de software libre, y se adentra, en parte, en la segunda condición. Para ello, se hará un exhaustivo estudio de las fuentes de datos para conocer el tipo de datos que se pueden extraer y los análisis que podemos realizar sobre los mismos. Posteriormente se analizarán los datos desde varias perspectivas, con especial hincapié en aquéllas que permitan conocer

mejor las novedosas formas de organizar los recursos humanos en el software libre, en especial en cuanto a auto-organización, distribución de tareas, creación de comunidades y estructura social del proyecto. Sin embargo, no se dejarán de lado aspectos más técnicos y que puedan aclarar el proceso de desarrollo de software libre; así, se reproducirán algunos de los estudios "clásicos" de ingeniería de software. También se estudiarán proyectos de software libre, en particular, y compilaciones de software libre, en general, para ver su evolución en el tiempo.

En la medida de lo posible, se intentará que el conocimiento adquirido -tanto el metodológico como el referente a los resultados-, pueda ser de utilidad para los propios proyectos de software libre y para el resto de la comunidad científica. Se buscará, por tanto, ofrecer los datos obtenidos de manera pública (preservando el anonimato de las personas involucradas si fuere necesario) y automatizar en la medida de lo posible los procesos de extracción y análisis. Para terminar, tanto la automatización como la experiencia adquirida se utilizarán para proponer una caracterización de proyectos de software libre basada en elementos cuantitativos.

C.3 Metodología

La metodología que se va a seguir en esta tesis se puede dividir en tres fases: una de identificación y descripción de las fuentes de datos, de los procesos de descarga, extracción, limpieza, almacenamiento e intercambio de los datos, una segunda de análisis de datos y finalmente una última donde se mostrarán algunos modelos y aplicaciones a partir de los datos obtenidos en las fases anteriores. Los análisis de los datos se han dividido a su vez en dos grupos: uno centrado en aspectos técnicos y otro en aspectos relacionados con los recursos humanos. A continuación, se describen todas las fases pormenorizadamente.

C.3.1 Fuentes de datos

Conocer a fondo las fuentes de datos disponibles es de vital importancia para la elaboración de estudios empíricos de proyectos de software libre. Esta primera fase abarca desde la identificación hasta el almacenamiento de los datos, discutiéndose al final la posibilidad de que todo este proceso se pueda realizar de manera automática por herramientas software. Las diferentes tareas que hay que realizar en cuanto a las fuentes de datos son:

- Identificación y descarga de datos de las fuentes

  Aún cuando las fuentes de datos son accesibles de manera pública (mayormente a través de Internet), han de ser identificadas. Las fuentes de datos se pueden encontrar de diferentes maneras y en diferentes lugares, de manera que se ha de escoger la mejor de ellas. Una vez que se ha hecho esto, se pasará a la descarga de datos que debería realizarse de manera no intrusiva para que el proyecto de software libre que esté siendo analizado no sufra efectos colaterales de la extracción y del análisis posterior (como, por ejemplo, sobrecarga de la máquina que hospeda el repositorio de versiones).

  Las fuentes de datos que se consideran en esta tesis son: código fuente, repositorios de versiones, listas de correo, sistemas de gestión de errores y, en menor medida, otras fuentes (repositorios de software, meta-datos e información de otra índole).

- Estudio de los datos y de su utilidad

  En general, las fuentes de datos se encuentran en un formato no estructurado o parcialmente estructurado que no ha sido pensado para ser analizado. En ocasiones incluso se hace necesario filtrar convenientemente la información deseada. Por ello, las fuentes de datos han de describirse de manera apropiada para poder ver la información que se puede extraer de las mismas y qué métodos son necesarios para tal fin.

- Extracción y limpieza de datos

  Una vez que se ha identificado y estudiado la fuente de datos, el siguiente paso incluye la extracción y limpieza de los datos. Los procedimientos que se han de considerar dependerán de

la fuente de datos y de la información que queremos obtener. Uno de los objetivos principales es extraer tanta información como sea posible aún cuando ésta pudiera no ser interesante para el análisis actual, porque probablemente lo serán en el futuro.

La limpieza de los datos es una tarea importante para algunas fuentes de datos, ya que los datos pueden no encontrarse en una manera apta para su análisis. Limpieza es un término muy amplio que incluye actividades como fusión (por ejemplo, si un desarrollador tiene diferentes nombres de usuario habrán de ser consideradas como una única) o agrupación (deberíamos, por ejemplo, identificar cambios que corresponden al código fuente y poder diferenciarlos de aquéllos que corresponden a comentarios).

Hay algunas fuentes para las que estos procesos están bien documentados, como en el caso del repositorio de versiones CVS, el más utilizado a día de hoy por proyectos de software libre [Zimmermann & Weissgerber, 2004]. Pero esto no es así para las demás fuentes que se consideran en esta tesis.

- Almacenamiento e intercambio de datos

  El almacenamiento de los datos en un formato conveniente es un paso importante para los análisis subsecuentes que queremos realizar sobre ellos. Dado que contamos con un gran volumen de datos, se utilizarán bases de datos relacionales por lo que habrá que crear estructuras relacionales que permitan la extracción de información de manera simple y, a ser posible, instantánea. Algunos de los análisis que se van a llevar a cabo, requieren estructuras de datos sofisticadas y peticiones a la base de datos optimizadas.

  Por otro lado, en no pocos casos a partir de los datos primarios obtenidos directamente de las fuentes de datos se pueden obtener más datos que enriquezcan el análisis. Un claro ejemplo de este enriquecimiento es la identificación de *commits* atómicos a partir de los commits en un repositorio de versiones CVS. Otro ejemplo sería la identificación y discriminación según el tipo de fichero (código fuente, documentación, traducción, etc.), lo que permitiría nuevos análisis como veremos más adelante.

  Algunas iniciativas más recientes en la comunidad científica de la ingeniería del software tratan de crear repositorios públicos que contengan datos del desarrollo de proyectos software [Sayyad Shirabad & Menzies, 2005]. Por eso, una de las metas será la posibilidad de poder exportar los datos. Esto permitirá a otros equipos de investigación reproducir y validar los resultados y, por supuesto, utilizar los datos para la realización de nuevos estudios con otros métodos. En este sentido, se presentarán varios formatos de intercambio como ARFF[3] u otros basados en XML.

- Automatización del proceso

  El hecho de que se pueden investigar multitud de proyectos con la misma metodología y que el volumen de datos sea tan grande, hace que la automatización de todo el proceso sea casi una necesidad. En muchos casos, esto viene facilitado porque muchos proyectos utilizan las mismas herramientas de ayuda al desarrollo (mismo sistema de control de versiones, mismo sistema de gestión de erratas...), por lo que una vez identificada la fuente de datos el resto de los procesos descritos anteriormente se pueden automatizar.

  Algunos grupos de investigación han creado y puesto a disposición pública las herramientas que han utilizado para sus análisis [Ghosh & Prakash, 2000; Wheeler, 2001; Germán, 2004c; Germán & Hindle, 2005; Mutton, 2003; Conklin *et al.*, 2005]; parte del trabajo de esta tesis será la creación de herramientas de apoyo a la extracción y el análisis de datos.

[3] ARFF (Attribute-Relation File Format) es un fichero de texto ASCII que describe una lista de instancias que comparten los mismos atributos. ARFF ha sido desarrollado para un proyecto de *machine learning* de la Universidad de Waikato en Nueva Zelanda. Más información en http://www.cs.waikato.ac.nz/ ml/weka/arff.html

C.3.2 Análisis de los datos

Mientras que las técnicas de adquisición y manipulación de datos presentadas anteriormente tienen una capacidad de mejora limitada, es en el campo del análisis donde las posibilidades son muy amplias. En esta tesis se mostrarán análisis desde dos perspectivas diferentes, aunque teniendo siempre en común su carácter empírico. Por un lado se presentarán análisis centrados en artefactos (el software en sí), denominado análisis técnico a partir de ahora. Por otro, se estudiará el desarrollo de software libre desde un punto de vista de los recursos humanos y de cómo están organizados.

Análisis técnico

El análisis técnico tiene como objetivo conocer mejor el producto software en sí, ver cómo evoluciona en el tiempo y tener una estimación de su calidad (o mantenibilidad). Se realizarán los siguientes análisis:

- Análisis básico de proyectos de software libre

  Un análisis básico permite obtener unos primeros resultados de un proyecto: su tamaño en términos de líneas de código fuente, el número de desarrolladores que han participado en su desarrollo, el número de *commits* en el sistema de control de versiones, el número de mensajes a las listas de correo, el número de erratas, etc. También es interesante mostrar la distribución de actividad entre los miembros del proyecto, ya que no todos los que colaboran lo hacen en la misma proporción. De esta forma, obtenemos unos primeros parámetros cuantitativos según los que caracterizar un proyecto de software libre.

- Evolución de software

  Debido a la disponibilidad en el tiempo del código fuente (ya sea a partir de las publicaciones de las diferentes versiones del código u obteniendo datos del sistema de control de versiones), podemos estudiar los proyectos de software libre desde una perspectiva de evolución del software. Como se ha comentado en los antecedentes, estudios recientes han mostrado que algunos proyectos de software libre parece que *infringen* las leyes de Lehman de evolución del software [Godfrey & Tu, 2000; 2001]. Uno de los objetivos de esta tesis es comprobar esta afirmación de modo general a partir del análisis de un número suficientemente grande de proyectos de software libre.

- Mantenimiento (arqueología) de software

  Según Parnas, hagamos lo que hagamos, el software envejece [Parnas, 1994]. Sin embargo, no se han realizado muchos esfuerzos para analizar y cuantificar de manera empírica este aspecto. El concepto de arqueología de software, similar al del *desgaste* del código [Eick *et al.*, 2001], nos permite ver la *edad* del software línea por línea, de manera que podamos determinar cuándo fue la última vez que se modificó una línea de código. Si se toman estas medidas a nivel de todo el software, podemos tener una idea cuantitativa de la *edad* del software y, bajo ciertas condiciones, lo que nos puede servir para conocer la mantenibilidad del mismo. Para ello, en esta tesis se proponen varias métricas que permiten comprobar el estado del software actual, que ayudan a conocer cuánto se ha mantenido el software y que pueden servir para estimar los costes del desarrollo/mantenimiento del software en el futuro.

- Evolución de compilaciones de software

  Aunque generalmente la evolución de software se ha centrado en proyectos aislados, la existencia de compilaciones de software en el mundo del software libre -las distribuciones- que incluyen cientos, a veces miles de programas, hace posible un análisis de evolución de software a este nivel. Así, se mostrarán los resultados de ver cómo cambia una distribución de software libre en el tiempo, entre otros en cuanto a su composición en número paquetes, de tamaño de los paquetes, de lenguajes de programación, etc. La finalidad de este estudio es conocer cómo evoluciona el software libre en el tiempo a partir de los programas incluidos en las distribuciones, que suelen ser a su vez los más demandados por los usuarios.

Análisis de recursos humanos y organización

Como se ha comentado con anterioridad, los recursos humanos con los que cuentan los proyectos de software libre son en su mayoría voluntarios. Esto abre nuevas incógnitas desde el punto de vista de ingeniería del software, ya que en entornos "clásicos", aún siendo difícilmente predecible la productividad de un grupo, siempre se sabía con el número de personas con el que se contaba así como el tiempo que dedicaban al proyecto. En los proyectos de software libre esto no es así; la predicción de tiempos de desarrollo y del número de personas dedicadas al proyecto es mucho más difícil de realizar, por lo que una estimación de costes se hace mucho más compleja. Por otro lado, las formas de gobierno y de organización de un proyecto de software libre son radicalmente diferentes y, en parte, desconocidas. En este sentido, se analizarán los siguientes aspectos:

- Distribución de tareas

  El desarrollo de software libre se basa fuertemente en la participación de voluntarios y, por tanto, el tiempo que estas personas dedican a un proyecto es difícil de estimar y puede variar de manera significativa entre unos y otros. Ha habido muchos esfuerzos dedicados a conocer cómo está estructurada la comunidad [Ghosh & Prakash, 2000; Mockus *et al.*, 2002; Koch & Schneider, 2002; Crowston *et al.*, 2003a; Crowston & Howison, 2005] que han demostrado que existe un grupo pequeño de desarrolladores que es responsable de una gran parte del trabajo. Se pretende profundizar en este área, mostrando que el núcleo se regenera con el tiempo, entrando nuevos desarrolladores a la vez que otros dejan de participar. Para ello se estudiará la composición del núcleo de desarrollo y se analizará cómo cambia en el tiempo.

- Integración de nuevos miembros

  El tiempo de integración que transcurre desde que alguien entra a colaborar en un proyecto de software libre hasta que termina siendo parte del núcleo del desarrollo del proyecto se ha de tener en cuenta. Por eso, se estudiará para aquéllos que han logrado entrar a formar parte del núcleo el tiempo desde las primeras contribuciones -generalmente pequeñas- que realizaron hasta que les fue permitido introducir cambios en el sistema de control de versiones. Este proceso se puede entender como uno en el que se va ganando paulatinamente la confianza del resto del equipo de desarrollo. Se comprobará a su vez si los patrones de integración de desarrolladores voluntarios y aquéllos que se dedican profesionalmente al desarrollo de una aplicación son diferentes o no.

- Análisis basado en tipo de fichero

  Generalmente, la investigación empírica de software libre ha puesto sus energías en el código fuente, entendiendo como tal artefactos escritos en un lenguaje de programación. Pero entre las fuentes de sistemas software modernos, y en especial entre aquéllos que tienen como objetivo a los usuarios finales, encontraremos además del código fuente muchos otros elementos (documentación, traducciones, ficheros de configuración de interfaz de usuarios, imágenes, multimedia, etc.). Éstos han de ser tratados durante el proceso de desarrollo, han de ser mantenidos y añaden complejidad a todo el sistema. Por supuesto, la existencia de diferentes tipos de ficheros es indicativa de diferentes tipos de actividades, por lo que es posible que haya especialización entre los integrantes del equipo de desarrollo.

  Como los proyectos de software libre no tienen una estructura jerárquica y están fuertemente ligados al desarrollo por parte de voluntarios, el desarrollo generalmente es auto-organizado y son los propios desarrolladores los que eligen las tareas que quieren realizar. Por eso, es interesante ver si se crean diferentes comunidades que trabajan en tipos de ficheros diferentes. Al disponer de datos en el tiempo, también se puede hacer este estudio con una perspectiva histórica.

  La metodología que se utilizará para este análisis parte de la discriminación de los ficheros según su tipo; la idea es identificar si un fichero incluye código fuente, documentación, traducciones, configuración de interfaz de usuario, etc. y analizar qué desarrolladores son los que interactúan con cada tipo de fichero.

- Análisis de redes sociales

El análisis de redes sociales es un conjunto de métodos que se pueden aplicar una vez definidas relaciones entre entidades. En nuestro caso, realizaremos un análisis de redes sociales para determinar la estructura social de los proyectos de software libre desde dos perspectivas: una teniendo en cuenta las personas que colaboran en los proyectos y sus interrelaciones, y otra que muestre las relaciones que existen entre los diferentes módulos -que generalmente se corresponden con aplicaciones- en un repositorio de control de versiones.

C.3.3 Resultados y modelos

Finalmente, en esta última parte de la tesis se utilizarán los datos obtenidos y analizados en las fases previas para la obtención de modelos de desarrollo o para otro tipo de aplicaciones. Se presentarán:

- Los resultados de aplicar las metodologías descritas a los casos de estudio

  Durante toda esta tesis, hemos aplicado las diferentes metodologías a los datos obtenidos de fuentes de datos de desarrollo públicas. En todos los casos, los proyectos que han sido estudiados son aplicaciones reales del mundo del software libre, por lo que se ha podido extraer conocimiento y hechos de interés y aplicación real.

- Un macro-modelo auto-organizativo del desarrollo

  Se presentará un modelo macroscópico de desarrollo de software libre. Se trata de un modelo macroscópico, ya que no intenta modelar el funcionamiento de un proyecto, sino más bien ayudar en la comprensión del fenómeno de software libre. El modelo está basado en agentes que trabajan en unos u otros proyectos de software libre según unos estímulos que perciben. Los estímulos son proporcionales al número de personas trabajando en un proyecto. Se presentará el algoritmo, se calibrará con datos de estudios empíricos de trabajos propios o publicados por otros grupos de investigación y se compararán los resultados obtenidos por medio de una simulación con los de otros trabajos de investigación.

C.4 Conclusiones

Entre las principales contribuciones de esta tesis podemos constatar que se trata del primer análisis exhaustivo de un gran número de proyectos software, aunque las metodologías propuestas y las herramientas que se han desarrollado para tal efecto permitan en un futuro próximo el estudio de todavía más proyectos. Gracias a la cantidad de datos que se puede recoger de manera pública, hemos podido reproducir algunos de los estudios clásicos de ingeniería del software -como es el caso del análisis de evolución- y aplicarlo a muchos proyectos de software libre.

Asimismo, se ha podido comprobar la importancia que tiene en la era de Internet complementar los análisis técnicos realizados sobre el producto con estudios socio-técnicos de las personas que están detrás del desarrollo del software.

Bibliography

[Albert & Barabasi, 2002] Reka Albert and Albert-Laszlo Barabasi. Statistical mechanics of complex networks. *Reviews of Modern Physics*, 74:47–97, 2002.

[Albert *et al.*, 2000] R. Albert, A. L. Barabasi, H. Jeong, and G. Bianconi. Power-law distribution of the world wide web. *Science*, 287, 2000.

[Amor *et al.*, 2005a] Juan José Amor, Jesús M. González-Barahona, Gregorio Robles, and Israel Herraiz. Measuring libre software using Debian 3.1 (sarge) as a case study: preliminary results. *Upgrade Magazine*, August 2005.

[Amor *et al.*, 2005b] Juan José Amor, Gregorio Robles, and Jesús M. González-Barahona. Measuring Woody: The size of Debian 3.0. Technical report, Grupo de Sistemas y Comunicaciones, Universidad Rey Juan Carlos, Madrid, Spain, June 2005.
http://gsyc.escet.urjc.es/publicaciones/tr/RoSaC-2005-10.pdf.

[Anthonisse, 1971] Jac M. Anthonisse. The rush in a directed graph. Technical report, Stichting Mathemastisch Centrum, Amsterdam, The Netherlands, 1971.

[Antoniades *et al.*, 2004] I.P. Antoniades, I. Samoladas, I. Stamelos, L. Aggelis, and G. L. Bleris. Dynamical simulation models of the open source development process. In Stefan Koch, editor, *Free/Open Source Software Development*, pages 174–202. Idea Group Publishing, Hershey, PA, 2004.

[Antoniol *et al.*, 2005] Giuliano Antoniol, Massimiliano Di Penta, Harald Gall, and Martin Pinzger. Towards the integration of versioning systems, bug reports and source code meta-models. *Electronic Notes in Theoretical Computer Science*, 127(3):87–99, April 2005.

[Arthur, 1988] L. Arthur. *Software Evolution: The Software Maintenance Challenge.* John Wiley and Sons, USA, 1988.

[Atkins *et al.*, 1999] David L. Atkins, Thomas Ball, Todd L. Graves, and Audris Mockus. Using version control data to evaluate the impact of software tools. In *Internatinal Conference on Software Engineering*, pages 324–333, Los Angeles, CA, USA, 1999.

[Atkins *et al.*, 2002] David L. Atkins, Thomas Ball, Todd L. Graves, and Audris Mockus. Using version control data to evaluate the impact of software tools: A case study of the version editor. *IEEE Transactions on Software Engineering*, 28(7):625–637, 2002.

[Beck, 1998] Kent Beck. *Extreme Programming Explained: Embrace Change.* Addison-Wesley Professional, 1998.

[Bezroukov, 1997] Nikolai Bezroukov. A second look at the cathedral and the bazar. *First Monday*, 4(12), 1997.

[Boehm *et al.*, 2000] Barry W. Boehm, Clark, Horowitz, Brown, Reifer, Chulani, Ray Madachy, and Bert Steece. *Software Cost Estimation with Cocomo II.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2000.

[Boehm, 1975] Barry B. Boehm. *The High Cost of Software. Practical Strategies for Development Large Software Systems.* Addison-Wesley Pub, 1975.

[Boehm, 1981] Barry B. Boehm. *Software Engineering Economics.* Prentice Hall, 1981.

[Bonabeau *et al.*, 1999] Eric Bonabeau, Marco Dorigo, and Guy Theraulaz. *Swarm Intelligence: From Natural to Articial Systems.* Oxford University Press, Inc., 1999.

[Brand, 2004] Andreas Brand. Ausführliche Fallstudie Open Source-Projekt KDE. Technical report, Johann Wolfgang Goethe-Universität, Frankfurt am Main, Germany, 2004.

[Brooks, 1987] Jr. Brooks, F.P. No silver bullet; essence and accidents of software engineering. *Computer*, 20(4):10–19, 1987.

[Brooks, 2001] Frederick P. Brooks. *The Mythical Man-Month.* Addison Wesley Longman Inc., New York, 2001.

[Burd & Munro, 1999] Elizabeth Burd and Malcolm Munro. Evaluating the evolution of a C application. In *Internation Workshop on Principles of Software Evolution*, Fukuoka, Japan, June 1999.

[Cancho & Sole, 2001] Cancho and R. Sole. The small world of human language. *Proceedings of the Royal Society of London. Series B, Biological Sciences*, 268:2261–2265, November 2001.

[Capiluppi *et al.*, 2003] Andrea Capiluppi, Patricia Lago, and Maurizio Morisio. Evidences in the evolution of OS projects through changelog analyses. In *Proceedings of the 3rd International Workshop on Open Source Software Engineering*, Orlando, Florida, USA, May 2003.

[Capiluppi *et al.*, 2004a] Andrea Capiluppi, Maurizio Morisio, and Patricia Lago. Evolution of understandability in OSS projects. In *Proceedings of the 8th European Conference on Software Maintenance and Reengineering*, Tampere, Finland, 2004.

[Capiluppi *et al.*, 2004b] Andrea Capiluppi, Maurizio Morisio, and Juan F. Ramil. Structural evolution of an Open Source system: a case study. In *Proceedings of the 12th International Workshop on Program Comprehension*, pages 172–183, Bari, Italy, 2004.

[Capiluppi, 2004] Andrea Capiluppi. Improving comprehension and cooperation through code structure. In *Proceedings of the 4th Workshop on Open Source Software Engineering, 26th International Conference on Software Engineering*, Edinburg, Scotland, UK, 2004.

[Celko, 2004] Joe Celko. *Joe Celko's Trees and Hierarchies in SQL for Smarties.* Morgan Kaufmann, 2004.

[Challet & Du, 2003] Damien Challet and Yann Le Du. Closed source versus Open Source in a model of software bug dynamics. http://arxiv.org/abs/cond-mat/0306511, 2003.

[Chen *et al.*, 2004] Kai Chen, Stephen R. Schach, Liguo Yu, A. Jefferson Offutt, and Gillian Z. Heller. Open-source change logs. *Empirical Software Engineering*, 9(3):197–210, 2004.

[Chidamber & Kemerer, 1994] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):474–493, June 1994.

[Conger, 1994] Sue A. Conger. *The New Software Engineering.* International Thomson Publishing, 1994.

[Conklin *et al.*, 2005] Megan Conklin, James Howison, and Kevin Crowston. Collaboration using OSSmole: A repository of FLOSS data and analyses. In *Proceedings of the International Workshop on Mining Software Repositories*, pages 126–130, St. Louis, Missouri, USA, May 2005.

[Conway, 1968] M.E. Conway. How do committees invent? *Datamation*, 14(4):28–31, 1968.

[Crowston & Howison, 2003] Kevin Crowston and James Howison. The social structure of open source software development teams. In *Proceedings of the International Conference on Information Systems*, Seattle, WA, USA, 2003.

[Crowston & Howison, 2005] Kevin Crowston and James Howison. The social structure of free and open source software development. *First Monday*, 10(2), February 2005. http://www.firstmonday.dk/issues/issue10_2/crowston/.

[Crowston *et al.*, 2003a] K. Crowston, B. Scozzi, and S. Buonocore. An explorative study of open source software development structure. In *Proceedings of the ECIS*, Naples, Italy, 2003.

[Crowston *et al.*, 2003b] Kevin Crowston, Hala Annabi, and James Howison. Defining open source software project success. In *Proceedings of 24th International Conference on Information Systems*, Seattle, WA, USA, December 2003.

[Daffara, 2002] Carlo Daffara. Open source as a different silver bullet. In *Workshop for Advancing the Research Agenda on Free/Open Source Software*, Brussels, Belgium, November 2002.

[Dalle & David, 2003] Jean-Michel Dalle and Paul A. David. The allocation of software development resources in Open Source production mode. Technical report, SIEPR Policy paper No. 02-027, SIEPR, Stanford, USA, 2003. http://siepr.stanford.edu/papers/pdf/02-27.pdf.

[David *et al.*, 2003] Paul A. David, Andrew Waterman, and Seema Arora. FLOSS-US the free/libre/open source software survey for 2003. Technical report, SIEPR Stanford Institute for Economic and Policy Research, Stanford, California, USA, 2003.

[Dekhtyar *et al.*, 2004] Alexander Dekhtyar, Jane Huffman Hayes, and Tim Menzies. Text is software too. In *Proceedings of the International Workshop on Mining Software Repositories*, Edinburg, Scotland, UK, 2004.

[Dempsey *et al.*, 1999] Bert J. Dempsey, Debra Weiss, Paul Jones, and Jane Greenberg. A quantitative profile of a community of open source linux developers, October 1999. http://www.ibiblio.org/osrt/develpro.html.

[Dinh-Trong & Bieman, 2004] Trungh Dinh-Trong and James M. Bieman. Open source software development: A case study of freebsd. In *Proceedings of the 10th International Software Metrics Symposium*, Chicago, IL, USA, 2004.

[Dinh-Trong & Bieman, 2005] Trung T. Dinh-Trong and James M. Bieman. The FreeBSD project: A replication case study of Open Source development. *IEEE Transactions on Software Engineering*, 31(6):481–494, June 2005.

[Ehrenkranzt, 2003] Justin R. Ehrenkranzt. Release management within open source projects. In *Proceedings of the 3rd International Workshop on Open Source Software Engineering*, Orlando, Florida, USA, 2003.

[Eick *et al.*, 2001] Stephen G. Eick, Todd L. Graves, Alan F. Karr, J. S. Marron, and Audris Mockus. Does code decay? Assessing the evidence from change management data. *IEEE Transactions on Software Engineering*, 27(1):1–12, 2001.

[Erenkrantz, 2003] Justin Erenkrantz. Release management within open source projects. In *Proceedings of the 3rd Workshop on Open Source Software Engineering, 25th International Conference on Software Engineering*, pages 51–55, Portland, Oregon, USA, 2003.

[Feller & Fitzgerald, 2002] Joseph Feller and Brian Fitzgerald. *Understanding Open Source Software Development*. Addison-Wesley, London, UK, 2002. Foreword by Eric S. Raymond. Companion Website.

[Fenton, 1991] N. Fenton. *Software Metrics: A Rigorous Approach.* Chapman and Hall, 1991.

[Ferenc *et al.*, 2004] Rudolf Ferenc, István Siket, and Tibor Gyimóthy. Extracting facts from Open Source software. In *Proceedings of the International Conference in Software Maintenance*, pages 60–69, Chicago, IL, USA, 2004.

[Fichman & Kemerer, 1997] Robert G. Fichman and Chris F. Kemerer. Object technology and reuse: Lessons from early adopters. *Computer*, 30(10):47–59, 1997.

[Fischer & Gall, 2004] Michael Fischer and Harald Gall. Visualizing feature evolution of large-scale software based on problem and modification report data. *Journal of Software Maintenance and Evolution: Research and Practice*, 16:385–403, 2004.

[Fischer *et al.*, 2003] Michael Fischer, Martin Pinzger, and Harald Gall. Populating a release history database from version control and bug tracking systems. In *Proceedings of the International Conference on Software Maintenance*, pages 23–32, Amsterdam, The Netherlands, September 2003.

[Fischer *et al.*, 2005] Michael Fischer, Johann Oberleitner, Jacek Ratzinger, and Harald Gall. Mining evolution data of a product family. In *MSR '05: Proceedings of the 2005 workshop on Mining software repositories*, pages 1–5, 2005.

[Frederick P. Brooks, 1978] Jr. Frederick P. Brooks. *The Mythical Man-Month: Essays on Softw.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1978.

[Freeman, 1977] Linton C. Freeman. A set of measures of centrality based on betweenness. *Sociometry 40, 35-41*, 1977.

[Fuggetta, 2003] Alfonso Fuggetta. Open source software - an evaluation. *Journal of Systems and Software*, 66(1):77–90, 2003.

[Gall *et al.*, 1997] Harald Gall, Mehdi Jazayeri, René Klösch, and Georg Trausmuth. Software evolution observations based on product release history. In *Proceedings of the International Conference on Software Maintenance*, pages 160–170. IEEE Computer Society, 1997.

[Gasser *et al.*, 2004] Les Gasser, Gabriel Ripoche, and Robert Sandusky. Research infrastructure for empirical science of foss. In *Proceedings of the International Workshop on Mining Software Repositories*, Edinburg, Scotland, UK, 2004.

[Germán & Hindle, 2005] Daniel M. Germán and Abram Hindle. Visualizing the evolution of software using softChange. *Journal of Software Engineering Knowledge Engineering*, 2005. Accepted for publication, under revisions.

[Germán & Mockus, 2003] Daniel M. Germán and Audris Mockus. Automating the measurement of open source projects. In *Proceedings of the 3rd Workshop on Open Source Software Engineering*, Portland, Oregon, USA, 2003.

[Germán, 2004a] Daniel M. Germán. An empirical study of fine-grained software modifications. In *Proceedings of the International Conference in Software Maintenance*, Chicago, IL, USA, 2004.

[Germán, 2004b] Daniel M. Germán. The GNOME project: a case study of open source, global software development. *Journal of Software Process: Improvement and Practice*, 8(4):201–215, 2004.

[Germán, 2004c] Daniel M. Germán. Mining CVS repositories, the softchange experience. In *Proceedings of the International Workshop on Mining Software Repositories*, pages 17–21, Edinburg, Scotland, UK, 2004.

[Germán, 2004d] Daniel M. Germán. Mining CVS repositories, the softChange experience. In *Proceedings of the International Workshop on Mining Software Repositories*, Edinburgh, UK, 2004.

[Germán, 2004e] Daniel M. Germán. Using software trails to reconstruct the evolution of software. *Journal of Software Maintenance and Evolution: Research and Practice*, 16(6):367–384, 2004.

[Ghosh & Prakash, 2000] Rishab A. Ghosh and Vipul Ved Prakash. The orbiten free software survey. *First Monday*, 5(7), May 2000.
http://www.firstmonday.dk/issues/issue5_7/ghosh/.

[Ghosh *et al.*, 2002a] Rishab Aiyer Ghosh, Ruediger Glott, Bernhard Krieger, and Gregorio Robles. Survey of developers (free/libre and open source software: Survey and study). Technical report, International Institute of Infonomics. University of Maastricht, The Netherlands, June 2002.
http://www.infonomics.nl/FLOSS/report.

[Ghosh *et al.*, 2002b] Rishab Aiyer Ghosh, Gregorio Robles, and Ruediger Glott. Software source code survey (free/libre and open source software: Survey and study). Technical report, International Institute of Infonomics. University of Maastricht, The Netherlands, June 2002.
http://www.infonomics.nl/FLOSS/report.

[Ghosh *et al.*, 2002c] Rishab Aiyer Ghosh, Gregorio Robles, and Ruediger Glott. Survey of developers - annexure on validation and methodology. Technical report, International Institute of Infonomics. University of Maastricht, The Netherlands, June 2002.
http://www.infonomics.nl/FLOSS/report.

[Gini, 1936] Conrado Gini. *On the Measure of Concentration with Espacial Reference to Income and Wealth*. Cowles Commission, 1936.

[Gîrba *et al.*, 2005] Tudor Gîrba, Adrian Kuhn, Mauricio Seeberger, and Stephane Ducasse. How developers drive software evolution. In *Proceedings of the International Workshop on Principles in Software Evolution*, pages 113–122, Lisboa, Portugal, September 2005.

[Girvan & Newman, 2002] M. Girvan and M.E.J. Newman. Community structure in social and biological networks. *Proc. Natl Acad. Scie. USA 99, 7821-7826*, 2002.

[Godfrey & Tu, 2000] Michael W. Godfrey and Quiang Tu. Evolution in Open Source software: A case study. In *Proceedings of the International Conference on Software Maintenance*, pages 131–142, San Jose, California, 2000.

[Godfrey & Tu, 2001] Michael Godfrey and Qiang Tu. Growth, evolution, and structural change in open source software. In *Internation Workshop on Principles of Software Evolution*, Vienna, Austria, September 2001.

[González-Barahona *et al.*, 2001] Jesús M. González-Barahona, Miguel A. Ortuño Pérez, Pedro de las Heras Quiros, José Centeno González, and Vicente Matellán Olivera. Counting potatoes: the size of Debian 2.2. *Upgrade Magazine*, II(6):60–66, December 2001.

[González-Barahona *et al.*, 2004]
Jesús M. González-Barahona, Gregorio Robles, Miguel Ortuño Pérez, Luis Rodero-Merino, José Centeno González, Vicente Matellan-Olivera, Eva Castro-Barbero, and Pedro de-las Heras-Quirós. Analyzing the anatomy of GNU/Linux distributions: methodology and case studies (Red Hat and Debian). In Stefan Koch, editor, *Free/Open Source Software Development*, pages 27–58. Idea Group Publishing, Hershey, Pennsylvania, USA, 2004.

[Grassé, 1959] Pierre-Paul Grassé. La reconstruction du nid et les coordinations inter-individuelles chez bellicositermes natalensis et cubitermes sp. la théorie de la stigmergie: Essai d'interpretation du comportement des termites constructeurs. *Insectes Sociaux*, (6):41–81, 1959.

[Graves & Mockus, 1998] Todd L. Graves and Audris Mockus. Inferring change effort from configuration management databases. In *5th IEEE International Software Metrics Symposium*, pages 267–, Bethesda, Maryland, USA, 1998.

[Guimera *et al.*, 2002] Roger Guimera, Albert Diaz-Guilera, F. Vega-Redondo, A. Cabrales, and Alex Arenas. Optimal network topologies for local search with congestion. *Physical Review Let. 89, 248701*, 2002.

[Hahsler & Koch, 2005] Matthias Hahsler and Stefan Koch. Discussion of a large-scale open source data collection methodology. In *Proceedings of the Hawaii International Conference on System Sciences (HICSS-38)*, Big Island, Hawaii, USA, January 2005.

[Hahsler, 2004] Michael Hahsler. A quantitative study of the adoption of design patterns by open source software developers. In Stefan Koch, editor, *Free/Open Source Software Development*, pages 103–123. Idea Group Publishing, Hershey, PA, 2004.

[Halstead, 1977] Maurice H. Halstead. *Elements of Software Science*. Elsevier, New York, USA, 1977.

[Healy & Schussman, 2003] Kieran Healy and Alan Schussman. The ecology of open-source software development. Technical report, University of Arizona, USA, January 2003. http://opensource.mit.edu/papers/healyschussman.pdf.

[Henry *et al.*, 1981] Sallie Henry, Dennis Kafura, and Kathy Harris. On the relationships among three software metrics. In *Proceedings of the 1981 ACM workshop/symposium on Measurement and evaluation of software quality*, pages 81–88, New York, NY, USA, 1981. ACM Press.

[Herbsleb *et al.*, 2001] James D. Herbsleb, Audris Mockus, Thomas A. Finholt, and Rebecca E. Grinter. An empirical study of global software development: distance and speed. In *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, pages 81–90, 2001.

[Hertel *et al.*, 2003] Guido Hertel, Sven Niedner, and Stefanie Herrmann. Motivation of software developers in open source projects: an Internet-based survey of contributors to the Linux kernel. *Research Policy*, 32(7):1159–1177, 2003.

[Hitz & Montazeri, 1996] Martin Hitz and Behzad Montazeri. Chidamber and kemerer's metrics suite: A measurement theory perspective. *IEEE Transactions on Software Engineering*, 22(4):267–271, 1996.

[Howison & Crowston, 2004] James Howison and Kevin Crowston. The perils and pitfalls of mining SourceForge. In *Proceedings of the International Workshop on Mining Software Repositories*, pages 7–11, Edinburg, Scotland, UK, 2004.

[Hunt & Johnson, 2002] F. Hunt and P. Johnson. On the Pareto distribution of Open Source projects. In *Proceedings of Open Source Software Development Workshop*, Newcastle, UK, 2002.

[Hunt & Thomas, 2002] Andy Hunt and Dave Thomas. Software Archaeology. *IEEE Software*, 19(2):20–22, March/April 2002.

[Iannacci, 2003] Federico Iannacci. The Linux managing model. *First Monday*, 8(12), 2003. http://www.firstmonday.dk/issues/issue8_12/iannacci/.

[Jensen & Scacchi, 2005] Chris Jensen and Walter Scacchi. Modeling recruitment and role migration processes in OSSD projects. In *Proceedings of 6th International Workshop on Software Process Simulation and Modeling*, St. Louis, May 2005.

[Koch & Schneider, 2002] Stefan Koch and Georg Schneider. Effort, cooperation and coordination in an open source software project: GNOME. *Information Systems Journal*, 12(1):27–42, 2002.

[Koch, 2005] Stefan Koch. Evolution of Open Source Software systems - a large-scale investigation. In *Proceedings of the 1st International Conference on Open Source Systems*, Genova, Italy, July 2005.

[Kohonen, 1990] Teuvo Kohonen. The self-organizing map. *Proc. IEEE*, 78:1464–1480, 1990.

[Kohonen, 1995] Teuvo Kohonen. *Self-Organizing Maps*. Springer, Berlin, Heidelberg, 1995.

[Krishnamurthy, 2002] Sandeep Krishnamurthy. Cave or community? An empirical examination of 100 mature Open Source projects. *First Monday*, 7(6), 2002.

[Kumar *et al.*, 2002] R. Kumar, P. Raghavan, S. Rajagopalan, and A. Tomkins. The web and social networks. *IEEE Computer*, 35(11):32–36, 2002.

[Lanzara & Morner, 2003] Giovan Francesco Lanzara and Michele Morner. The knowledge ecology of open-source software projects. In *Proceedings of the 19th EGOS (European Group of Organizational Studies) Colloquim*, July 2003.

[Latora & Marchiori, 2003] Vito Latora and Massimo Marchiori. Economic small-world behavior in weighted networks. *Euro Physics Journal B 32, 249-263*, 2003.

[Lehman & Belady, 1985] Manny M. Lehman and L. A. Belady, editors. *Program evolution: Processes of software change*. Academic Press Professional, Inc., San Diego, CA, USA, 1985.

[Lehman & Ramil, 2001] Manny M. Lehman and Juan F. Ramil. Rules and tools for software evolution planning and management. *Annals of Software Engineering*, 11(1):15–44, 2001.

[Lehman *et al.*, 1997] Manny M. Lehman, Juan F. Ramil, P D. Wernick, D E. Perry, and W M. Turski. Metrics and laws of software evolution - the nineties view. In *METRICS '97: Proceedings of the 4th International Symposium on Software Metrics*, page 20, nov 1997.

[Lehman *et al.*, 2001] Manny M. Lehman, Juan F. Ramil, and U. Sandler. An approach to modelling long-term growth trends in software systems. In *Internation Conference on Software Maintenance*, pages 219–228, Florence, Italy, November 2001.

[López & Sanjuan, 2002] Luis López and Miguel Angel Fernandez Sanjuan. Relation between structure and size in social networks. *Physical Review Let. 89, 248701*, 2002.

[López *et al.*, 2002] Luis López, J.F. Mendes, and Miguel Ángel F. Sanjuan. Hierarchical social networkds and information flow. *Physica A, 316, 591-604*, 2002.

[Madey *et al.*, 2002] Gregory Madey, Vincent Freeh, and Renee Tynan. The open source development phenomenon: An analysis based on social network theory. In *Americas Conference on Information Systems (AMCIS2002)*, pages 1806–1813, Dallas, Texas, USA, 2002.
http://www.nd.edu/ oss/Papers/amcis_oss.pdf.

[Madey *et al.*, 2004] Gregory Madey, Vincent Freeh, and Renee Tynan. Modeling the Free/Open Source software community: A quantitative investigation. In Stefan Koch, editor, *Free/Open Source Software Development*, pages 203–221. Idea Group Publishing, Hershey, Pennsylvania, USA, 2004.

[Massey, 2005] Bart Massey. Longitudinal analysis of long-timescale open source repository data. In *Proceedings of the International Workshop on Predictor Models in Software Engineering (PROMISE 2005)*, St.Louis, Missouri, USA, 2005.

[McCabe, 1976] Thomas J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, December 1976.

[Michlmayr, 2004] Martin Michlmayr. Managing volunteer activity in free software projects. In *Proceedings of the USENIX 2004 Annual Technical Conference, FREENIX Track*, pages 93–102, Boston, USA, 2004.

[Minkov *et al.*, 2005] Einat Minkov, Richard Wang, and William Cohen. Extracting personal names from emails: Applying named entity recognition to informal text. In *Proceedings of the Human Language Technology Conference. Conference on Empirical Methods in Natural Language Processing*, Vancouver, B.C., Canada, October 2005.

[Mockus & Votta, 2000] Audris Mockus and Lawrence G. Votta. Identifying reasons for software changes using historic databases. In *Proceedings of the International Conference on Software Maintenance*, pages 120–130, October 2000.

[Mockus *et al.*, 2002] Audris Mockus, Roy T. Fielding, and James D. Herbsleb. Two case studies of Open Source software development: Apache and Mozilla. *ACM Transactions on Software Engineering and Methodology*, 11(3):309–346, 2002.

[Mutton, 2003] Paul Mutton. Inferring and visualizing social networks on irc. In *Eighth International Conference on Information Visualization (IV04)*, July 2003.

[Newman, 2001a] Mark E. J. Newman. Scientific collaboration networks: I. network construction and fundamental results. *Phys. Rev. E 64, 016131*, 2001.

[Newman, 2001b] Mark E. J. Newman. Scientific collaboration networks: Ii. shortest paths, weighted networks, and centrality. *Phys. Rev. E 64, 016132*, 2001.

[Norden, 1963] Peter V. Norden. Useful tools for project management. *Operations Research in Research and Development*, 1963.

[Ohira *et al.*, 2005] Masao Ohira, Naoki Ohsugi, Tetsuya Ohoka, and Ken ichi Matsumoto. Accelerating cross-project knowledge collaboration using collaborative filtering and social networks. In *Proceedings of the International Workshop on Mining Software Repositories*, St. Louis, Missouri, USA, May 2005.

[O'Mahoney & Ferraro, 2004] Siobhán O'Mahoney and Fabrizio Ferraro. Managing the boundary of an 'open' project. Technical Report D/537, IESE Business School, January 2004. available at http://ideas.repec.org/p/ebg/iesewp/d-0537.html.

[Oman & Hagemeister, 1992] Paul Oman and Jack Hagemeister. Metrics for assessing a software system's maintainability. In *International Conference on Software Maintenance*, pages 337–344, Los Alamitos, CA, 1992. IEEE Computer Society Press.

[Oman & Hagemeister, 1994] Paul Oman and Jack Hagemeister. Construction and testing of polynomials predicting software maintainability. *Journal of Systems and Software*, 24(3), March 1994.

[Parnas, 1972] David L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15:1053–1058, 1972.

[Parnas, 1994] David Lorge Parnas. Software aging. In *Proceedings of the International Conference on Software Engineering*, pages 279–287, Sorrento, Italy, May 1994.

[Paulson *et al.*, 2004] J. W. Paulson, Giancarlo Succi, and A. Eberlein. An empirical study of open-source and closed-source software products. *Transactions on Software Engineering*, 30(4), April 2004.

[Perens, 1999] Bruce Perens. The open source definition. In Chris DiBona, Sam Ockman, and Mark Stone, editors, *Open Sources: Voices from the Open Source Revolution*. O'Reilly and Associates, Cambridge, Massachusetts, 1999.

[Putnam, 1978] Lawrence H. Putnam. A general empirical solution to the macro software sizing and estimating problem. *IEEE Transactions on Software Engineering, 4: 345-361*, 1978.

[Ramil & Lehman, 2000a] Juan F. Ramil and Meir M. Lehman. Effort estimation from change records of evolving software (poster session). In *Proceedings of the 22nd international conference on Software engineering*, page 777. ACM Press, 2000.

[Ramil & Lehman, 2000b] Juan F. Ramil and Meir M. Lehman. Metrics of software evolution as effort predictors - a case study. In *Proceedings of the International Conference on Software Maintenance*, pages 163–172. IEEE Computer Society, 2000.

[Raymond, 1998] Eric S. Raymond. The cathedral and the bazar. *First Monday*, 3(3), March 1998. http://www.firstmonday.dk/issues/issue3_3/raymond/.

[Robles *et al.*, 2001] Gregorio Robles, Hendrik Scheider, Ingo Tretkowski, and Niels Weber. Who is doing it? A research on libre software developers. Technical report, Fachgebiet für Informatik und Gesellschaft, Technische Universität Berlin, Berlin, Germany, August 2001. http://widi.berlios.de/paper/study.html.

[Royce, 1970] Winston Royce. Managing the development of large software system. In *Proceedings of the IEEE WESCON*, pages 1–9, August 1970.

[Rysselberghe & Demeyer, 2004] Filip Van Rysselberghe and Serge Demeyer. Studying software evolution information by visualizing the change history. In *International Conference on Software Maintenance*, pages 328–337, 2004.

[Sabidussi, 1996] Gert Sabidussi. The centrality index of a graph. *Psychometirka 31, 581-606*, 1996.

[Salus, 2005] Peter H. Salus. The daemon, the gnu and the penguin. Published as a series of articles in Groklaw, 2005. http://www.groklaw.net/article.php?story=20050623114426823.

[Samoladas *et al.*, 2004] Ioannis Samoladas, Ioannis Stamelos, Lefteris Angelis, and Apostolos Oikonomou. Open source software development should strive for even greater code maintainability. *Communications of the ACM*, 47(10), October 2004.

[Sayyad Shirabad & Menzies, 2005] Jelber Sayyad Shirabad and Tim J. Menzies. The PROMISE Repository of Software Engineering Databases. School of Information Technology and Engineering, University of Ottawa, Canada, 2005.

[Sayyad Shirabad, 2004] Jelber Sayyad Shirabad. *Supporting Software Maintenance by Mining Software Update Records*. PhD thesis, University of Ottawa, May 2004.

[Scacchi, 2004] Walt Scacchi. Free and Open Source development practices in the game community. *IEEE Software*, 21(1):59–66, 2004.

[Schach *et al.*, 2002] Stephen R. Schach, Bo Jin, David R. Wright, Gillian Z. Heller, and A. Jefferson Offutt. Maintainability of the Linux kernel. *IEE Proceedings–Software*, 149:18–23, 2002.

[Schach, 2002] Steven R. Schach. *Object-Oriented and Classical Software Engineering*. McGraw-Hill Companies, Inc., 2002.

[Senyard & Michlmayr, 2004] Anthony Senyard and Martin Michlmayr. How to have a successful free software project. In *Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC'04)*, pages 84–91, Busan, Korea, 2004. IEEE Computer Society.

[Spinellis, 2003] Diomidis Spinellis. *Code Reading: The Open Source Perspective*. Addison Wesley Professional, 2003.

[Stallman *et al.*, 2002] Richard M. Stallman, Lawrence Lessig, and Joshua Gay. *Free Software, Free Society: Selected Essays of Richard M. Stallman*. Free Software Foundation, October 2002.

[Stallman, 1999] Richard Stallman. The GNU operating system and the free software movement. In Chris DiBona, Sam Ockman, and Mark Stone, editors, *Open Sources: Voices from the Open Source Revolution*. O'Reilly and Associates, Cambridge, Massachusetts, 1999.

[Succi *et al.*, 2001] Giancarlo Succi, J. W. Paulson, and A. Eberlein. Preliminary results from an empirical study on the growth of open source and commercial software products. In *EDSER-3 Workshop, co-located with ICSE 2001*, Toronto, Canada, May 2001.

[Swanson, 1976] E. Burton Swanson. The dimensions of maintenance. In *Proceedings of the 2nd International conference on Software Engineering*, pages 492–497. IEEE Computer Society Press, 1976.

[Tuomi, 2004] Ilkka Tuomi. Evolution of the Linux Credits file: Methodological challenges and reference data for Open Source research. *First Monday*, 9(6), 2004.
http://www.firstmonday.dk/issues/issue9_6/ghosh/.

[Turski, 1996] Wladyslaw M. Turski. Reference model for smooth growth of software systems. *IEEE Transactions on Software Engineering*, 22(8):599–600, 1996.

[Ultsch & Siemon, 1990] A. Ultsch and H.P. Siemon. Kohonen's self organizing feature maps for exploratory data analysis. In *Proc. INNC'90, Int. Neural Network Conf.*, pages 305–308, Dordrecht, Netherlands, 1990. Kluwer.

[Villa, 2003] Luis Villa. How gnome learned to stop worrying and love the bug. In *Talk at the Otawa Linux Symposium*, Otawa, July 2003.
http://tieguy.org/talks/OLS-2003-html/.

[Villa, 2005] Luis Villa. Why everyone needs a bugmaster. In *Talk at linux.conf.au*, Canberra, April 2005.
http://tieguy.org/talks/LCA-2005-paper-html/.

[Vixie, 1999] Paul Vixie. Open source software engineering. In Chris DiBona, Sam Ockman, and Mark Stone, editors, *Open Sources: Voices from the Open Source Revolution*. O'Reilly and Associates, Cambridge, Massachusetts, 1999.

[von Krogh *et al.*, 2003] Georg von Krogh, Sebastian Spaeth, and Karim R. Lakhani. Community, joining, and specialization in Open Source Software innovation: A case study. *MIT Sloan Working Paper No. 4413-03*, 2003.

[Watts & Strogatz, 1998] Duncan J. Watts and S.H. Strogatz. Collective dynamics of small-world networks. *Nature 393, 440-442*, 1998.

[Watts, 1999] Duncan J. Watts. *Small Worlds*. Princeton Univesity Press, 1999.

[Watts, 2003] Duncan J. Watts. *Six Degrees*. W. W. Norton & Company, New York, 2003.

[Weinberg, 1998] Gerald M. Weinberg. *The Psychology of Computer Programming: Silver Anniversary Edition*. Dorset House Publishing Company, Incorporated, 1998.

[Wheeler, 2000] David A. Wheeler. Estimating linux's size, November 2000.
http://www.dwheeler.com/sloc/redhat62-v1/redhat62sloc.html.

[Wheeler, 2001] David A. Wheeler. More than a gigabuck: Estimating GNU/Linux's size, June 2001.
http://www.dwheeler.com/sloc/redhat71-v1/redhat71sloc.html.

[Wheeler, 2004] David A. Wheeler. Linux kernel 2.6: It's worth more!, October 2004.
http://www.dwheeler.com/essays/linux-kernel-cost.html.

[Williams, 2002] Sam Williams. *Free as in Freedom. Richard Stallman's Crusade for Free Software*. O'Reilly, March 2002.

[Yamamoto *et al.*, 2005] Tetsuo Yamamoto, Makoto Matsushita, Toshihiro Kamiya, and Katsuro Inoue. Measuring similarity of large software systems based on source code correspondence. In *6th International PROFES (Product Focused Software Process Improvement) conference, PROFES 2005*, Oulu, Finland, June 2005.

[Ye *et al.*, 2004] Yuwan Ye, Kumiyo Nakakoji, Yasuhiro Yamamoto, and Kouichi Kishida. The co-evolution of systems and communities in Free and Open Source software development. In Stefan Koch, editor, *Free/Open Source Software Development*, pages 59–82. Idea Group Publishing, Hershey, Pennsylvania, USA, 2004.

[Zimmermann & Weissgerber, 2004] Thomas Zimmermann and Peter Weissgerber. Processing CVS data for fine-grained analysis. In *Proceedings of the International Workshop on Mining Software Repositories*, Edinburg, Scotland, UK, 2004.

[Zimmermann *et al.*, 2005] Thomas Zimmermann, Peter Weißgerber, Stephan Diehl, and Andreas Zeller. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, 31(6):429–445, June 2005.

[Zuse, 1991] Horst Zuse. *Software Complexity: Measures and Methods*. DeGruyter Programming Complex Systems Series, New York, USA, 1991.

Appendix D

Creative Commons Attribution-ShareAlike 2.0

License

THE WORK (AS DEFINED BELOW) IS PROVIDED UNDER THE TERMS OF THIS CREATIVE COMMONS PUBLIC LICENSE ("CCPL" OR "LICENSE"). THE WORK IS PROTECTED BY COPYRIGHT AND/OR OTHER APPLICABLE LAW. ANY USE OF THE WORK OTHER THAN AS AUTHORIZED UNDER THIS LICENSE OR COPYRIGHT LAW IS PROHIBITED.

BY EXERCISING ANY RIGHTS TO THE WORK PROVIDED HERE, YOU ACCEPT AND AGREE TO BE BOUND BY THE TERMS OF THIS LICENSE. THE LICENSOR GRANTS YOU THE RIGHTS CONTAINED HERE IN CONSIDERATION OF YOUR ACCEPTANCE OF SUCH TERMS AND CONDITIONS.

1. Definitions

1. "**Collective Work**" means a work, such as a periodical issue, anthology or encyclopedia, in which the Work in its entirety in unmodified form, along with a number of other contributions, constituting separate and independent works in themselves, are assembled into a collective whole. A work that constitutes a Collective Work will not be considered a Derivative Work (as defined below) for the purposes of this License.

2. "**Derivative Work**" means a work based upon the Work or upon the Work and other pre-existing works, such as a translation, musical arrangement, dramatization, fictionalization, motion picture version, sound recording, art reproduction, abridgment, condensation, or any other form in which the Work may be recast, transformed, or adapted, except that a work that constitutes a Collective Work will not be considered a Derivative Work for the purpose of this License. For the avoidance of doubt, where the Work is a musical composition or sound recording, the synchronization of the Work in timed-relation with a moving image ("synching") will be considered a Derivative Work for the purpose of this License.

3. "**Licensor**" means the individual or entity that offers the Work under the terms of this License.

4. "**Original Author**" means the individual or entity who created the Work.

5. "**Work**" means the copyrightable work of authorship offered under the terms of this License.

6. "**You**" means an individual or entity exercising rights under this License who has not previously violated the terms of this License with respect to the Work, or who has received express permission from the Licensor to exercise rights under this License despite a previous violation.

7. "**License Elements**" means the following high-level license attributes as selected by Licensor and indicated in the title of this License: Attribution, ShareAlike.

2. Fair Use Rights. Nothing in this license is intended to reduce, limit, or restrict any rights arising from fair use, first sale or other limitations on the exclusive rights of the copyright owner under copyright law or other applicable laws.

3. License Grant. Subject to the terms and conditions of this License, Licensor hereby grants You a worldwide, royalty-free, non-exclusive, perpetual (for the duration of the applicable copyright) license to exercise the rights in the Work as stated below:

1. to reproduce the Work, to incorporate the Work into one or more Collective Works, and to reproduce the Work as incorporated in the Collective Works;
2. to create and reproduce Derivative Works;
3. to distribute copies or phonorecords of, display publicly, perform publicly, and perform publicly by means of a digital audio transmission the Work including as incorporated in Collective Works;
4. to distribute copies or phonorecords of, display publicly, perform publicly, and perform publicly by means of a digital audio transmission Derivative Works.
5. For the avoidance of doubt, where the work is a musical composition:
   (a) **Performance Royalties Under Blanket Licenses**. Licensor waives the exclusive right to collect, whether individually or via a performance rights society (e.g. ASCAP, BMI, SESAC), royalties for the public performance or public digital performance (e.g. webcast) of the Work.
   (b) **Mechanical Rights and Statutory Royalties**. Licensor waives the exclusive right to collect, whether individually or via a music rights society or designated agent (e.g. Harry Fox Agency), royalties for any phonorecord You create from the Work ("cover version") and distribute, subject to the compulsory license created by 17 USC Section 115 of the US Copyright Act (or the equivalent in other jurisdictions).
6. **Webcasting Rights and Statutory Royalties**. For the avoidance of doubt, where the Work is a sound recording, Licensor waives the exclusive right to collect, whether individually or via a performance-rights society (e.g. SoundExchange), royalties for the public digital performance (e.g. webcast) of the Work, subject to the compulsory license created by 17 USC Section 114 of the US Copyright Act (or the equivalent in other jurisdictions).

The above rights may be exercised in all media and formats whether now known or hereafter devised. The above rights include the right to make such modifications as are technically necessary to exercise the rights in other media and formats. All rights not expressly granted by Licensor are hereby reserved.

4. Restrictions. The license granted in Section 3 above is expressly made subject to and limited by the following restrictions:

1. You may distribute, publicly display, publicly perform, or publicly digitally perform the Work only under the terms of this License, and You must include a copy of, or the Uniform Resource Identifier for, this License with every copy or phonorecord of the Work You distribute, publicly display, publicly perform, or publicly digitally perform. You may not offer or impose any terms on the Work that alter or restrict the terms of this License or the recipients' exercise of the rights granted hereunder. You may not sublicense the Work. You must keep intact all notices that refer to this License and to the disclaimer of warranties. You may not distribute, publicly display, publicly perform, or publicly digitally perform the Work with any technological measures that control access or use of the Work in a manner inconsistent with the terms of this License Agreement. The above applies to the Work as incorporated in a Collective Work, but this does not require the Collective Work apart from the Work itself to be made subject to the terms of this License. If You create a Collective Work, upon notice from any Licensor You must, to the extent practicable, remove from the Collective Work any reference to such Licensor or the

Original Author, as requested. If You create a Derivative Work, upon notice from any Licensor You must, to the extent practicable, remove from the Derivative Work any reference to such Licensor or the Original Author, as requested.

2. You may distribute, publicly display, publicly perform, or publicly digitally perform a Derivative Work only under the terms of this License, a later version of this License with the same License Elements as this License, or a Creative Commons iCommons license that contains the same License Elements as this License (e.g. Attribution-ShareAlike 2.0 Japan). You must include a copy of, or the Uniform Resource Identifier for, this License or other license specified in the previous sentence with every copy or phonorecord of each Derivative Work You distribute, publicly display, publicly perform, or publicly digitally perform. You may not offer or impose any terms on the Derivative Works that alter or restrict the terms of this License or the recipients' exercise of the rights granted hereunder, and You must keep intact all notices that refer to this License and to the disclaimer of warranties. You may not distribute, publicly display, publicly perform, or publicly digitally perform the Derivative Work with any technological measures that control access or use of the Work in a manner inconsistent with the terms of this License Agreement. The above applies to the Derivative Work as incorporated in a Collective Work, but this does not require the Collective Work apart from the Derivative Work itself to be made subject to the terms of this License.

3. If you distribute, publicly display, publicly perform, or publicly digitally perform the Work or any Derivative Works or Collective Works, You must keep intact all copyright notices for the Work and give the Original Author credit reasonable to the medium or means You are utilizing by conveying the name (or pseudonym if applicable) of the Original Author if supplied; the title of the Work if supplied; to the extent reasonably practicable, the Uniform Resource Identifier, if any, that Licensor specifies to be associated with the Work, unless such URI does not refer to the copyright notice or licensing information for the Work; and in the case of a Derivative Work, a credit identifying the use of the Work in the Derivative Work (e.g., "French translation of the Work by Original Author," or "Screenplay based on original Work by Original Author"). Such credit may be implemented in any reasonable manner; provided, however, that in the case of a Derivative Work or Collective Work, at a minimum such credit will appear where any other comparable authorship credit appears and in a manner at least as prominent as such other comparable authorship credit.

5. Representations, Warranties and Disclaimer

UNLESS OTHERWISE AGREED TO BY THE PARTIES IN WRITING, LICENSOR OFFERS THE WORK AS-IS AND MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND CONCERNING THE MATERIALS, EXPRESS, IMPLIED, STATUTORY OR OTHERWISE, INCLUDING, WITHOUT LIMITATION, WARRANTIES OF TITLE, MERCHANTIBILITY, FITNESS FOR A PARTICULAR PURPOSE, NONINFRINGEMENT, OR THE ABSENCE OF LATENT OR OTHER DEFECTS, ACCURACY, OR THE PRESENCE OF ABSENCE OF ERRORS, WHETHER OR NOT DISCOVERABLE. SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO SUCH EXCLUSION MAY NOT APPLY TO YOU.

6. Limitation on Liability. EXCEPT TO THE EXTENT REQUIRED BY APPLICABLE LAW, IN NO EVENT WILL LICENSOR BE LIABLE TO YOU ON ANY LEGAL THEORY FOR ANY SPECIAL, INCIDENTAL, CONSEQUENTIAL, PUNITIVE OR EXEMPLARY DAMAGES ARISING OUT OF THIS LICENSE OR THE USE OF THE WORK, EVEN IF LICENSOR HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

7. Termination

1. This License and the rights granted hereunder will terminate automatically upon any breach by You of the terms of this License. Individuals or entities who have received Derivative Works or Collective Works from You under this License, however, will not have their licenses terminated

provided such individuals or entities remain in full compliance with those licenses. Sections 1, 2, 5, 6, 7, and 8 will survive any termination of this License.

2. Subject to the above terms and conditions, the license granted here is perpetual (for the duration of the applicable copyright in the Work). Notwithstanding the above, Licensor reserves the right to release the Work under different license terms or to stop distributing the Work at any time; provided, however that any such election will not serve to withdraw this License (or any other license that has been, or is required to be, granted under the terms of this License), and this License will continue in full force and effect unless terminated as stated above.

8. Miscellaneous

1. Each time You distribute or publicly digitally perform the Work or a Collective Work, the Licensor offers to the recipient a license to the Work on the same terms and conditions as the license granted to You under this License.

2. Each time You distribute or publicly digitally perform a Derivative Work, Licensor offers to the recipient a license to the original Work on the same terms and conditions as the license granted to You under this License.

3. If any provision of this License is invalid or unenforceable under applicable law, it shall not affect the validity or enforceability of the remainder of the terms of this License, and without further action by the parties to this agreement, such provision shall be reformed to the minimum extent necessary to make such provision valid and enforceable.

4. No term or provision of this License shall be deemed waived and no breach consented to unless such waiver or consent shall be in writing and signed by the party to be charged with such waiver or consent.

5. This License constitutes the entire agreement between the parties with respect to the Work licensed here. There are no understandings, agreements or representations with respect to the Work not specified here. Licensor shall not be bound by any additional provisions that may appear in any communication from You. This License may not be modified without the mutual written agreement of the Licensor and You.

Creative Commons is not a party to this License, and makes no warranty whatsoever in connection with the Work. Creative Commons will not be liable to You or any party on any legal theory for any damages whatsoever, including without limitation any general, special, incidental or consequential damages arising in connection to this license. Notwithstanding the foregoing two (2) sentences, if Creative Commons has expressly identified itself as the Licensor hereunder, it shall have all rights and obligations of Licensor.

Except for the limited purpose of indicating to the public that the Work is licensed under the CCPL, neither party will use the trademark "Creative Commons" or any related trademark or logo of Creative Commons without the prior written consent of Creative Commons. Any permitted use will be in compliance with Creative Commons' then-current trademark usage guidelines, as may be published on its website or otherwise made available upon request from time to time.

Creative Commons may be contacted at http://creativecommons.org/.