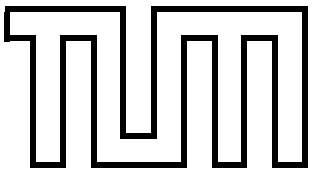
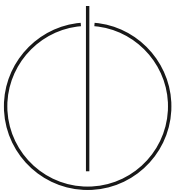


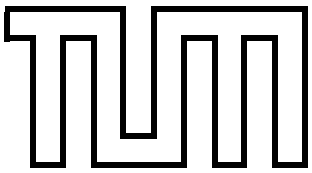
FAKULTÄT FÜR INFORMATIK
TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

TYPICAL DEVELOPMENT PROCESSES OF FREE AND OPEN SOURCE SOFTWARE PROJECTS

DANIEL G. SIEGEL



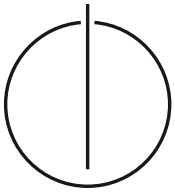


FAKULTÄT FÜR INFORMATIK
Technische Universität München

Master's Thesis in Informatics

TYPICAL DEVELOPMENT PROCESSES OF FREE AND OPEN SOURCE SOFTWARE PROJECTS

AUTHOR: DANIEL G. SIEGEL
SUPERVISOR: UNIV.-PROF. DR. PETER HUBWIESER
ADVISOR: MARCUS BITZL
DATE: MAY 15TH, 2012



DECLARATION

I assure the single handed composition of this master's thesis only supported by declared resources.

Munich, May 15th, 2012

Daniel G. Siegel

ABSTRACT

Free and Open Source Software is a research subject which has constantly been gaining importance for modern digital life over the past 20 years. The World Wide Web as we know it today would have been impossible without the free availability of source code, vivid development communities and consequent support for open standards. Despite these facts, there has been little research about the particular similarities or differences in the development processes and structures of Free and Open Source Software projects.

This thesis aims to provide an introduction into the development processes and project management strategies of Free and Open Source Software projects. To accomplish this, a project analysis catalogue is established with which Free and Open Source Software can be analyzed systematically and satisfactorily for the goals of this thesis. Following this catalogue several Free and Open Source Software projects are analyzed in order to identify concertedly models and processes.

The results corroborate suggestions that especially large and mature Free and Open Source Software projects have established similar structures and processes. This includes very hierarchical structures with akin roles, similar release cycles and analogical development processes which cannot be resembled with traditional or agile software engineering methods.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

ABI	Application Binary Interface
API	Application Programming Interface
BDFL	Benevolent Dictator For Life
CDE	Common Desktop Environment
CMF	Content Management Framework
CMS	Content Management System
DFSG	Debian Free Software Guidelines
FESCo	Fedora Engineering Steering Committee
FOSS	Free and Open Source Software
GNU	GNU's Not Unix
GNU GPL	GNU General Public License
GNU LGPL	GNU Lesser General Public License
GTK	GIMP Tool Kit
IRC	Internet Relay Chat
KDE SC	KDE Software Compilation
ORDBMS	Object-Relational Database Management System
PEP	Python Enhancement Proposal
PLIP	Plone Improvement Proposal
QPL	Q Public License
RFC	Request for Comments
SIG	Special Interest Group
SQL	Structured Query Language
UI	User Interface
UTC	Coordinated Universal Time
XP	Extreme Programming

1 INTRODUCTION

1.1 PROBLEM AND MOTIVATION

Without Free and Open Source Software (FOSS) the modern digital life would be different. FOSS projects empower *Fortune 500* companies such as Google, Amazon, Red Hat or Facebook. They also are key figures in the internet, running on most of the servers, but also on smartphones or other devices. However FOSS projects do not only have an important role in the modern software ecosystem, there is also an active development community behind them which consists of volunteers but also companies.

Lowdown such as the costs, the free availability of the source code and the number of people who work on a single project are certainly positive aspects of FOSS projects. They allow to use them without big investments and risks to implement an idea quickly and effectively. A matching example is the company Google which used the Linux operating system to build big data centers cost-effectively. That is just one of the many reasons why FOSS has become quite popular especially in the last years.

On account of the different origins and backgrounds of the single projects, different development models have evolved, which seem inimitable. Examples are the different communities, development and release cycles, used tools or the project structure. When comparing them at first glance they indeed look quite different, too diverse seem the different project leaders, the communities and the goals of each project. The question here is now whether projects have common grounds or use so called *best practices*.

Unfortunately there are not too many scrutinies which aim for similarities or differences in the development processes of FOSS projects. There exist however many studies about single aspects of projects, such as the motivation of developers, the social structure inside a project, the communication, development workflows or software engineering methods. This thesis aims to provide a primary step into this direction, summarizing the findings and trying to establish a common ground.

Concretely, this study tries to answer the following research question: How do FOSS projects work, which structures do they have and which workflows have they established. To accomplish this, several FOSS will be analyzed in order to identify concertedly models. In addition they will be compared to traditional software engineering models in order to see whether they are similar or oppose differences.

1.2 OUTLINE OF THE THESIS

CHAPTER 1 – INTRODUCTION This chapter presents an overview of the thesis and introduces the reader to the problem and motivation of this analysis.

CHAPTER 2 – THEORETICAL BACKGROUND In order to empower a scientific research for the project analysis, several research methods and tools will be introduced in this chapter. Additionally traditional and agile software engineering methods will be discussed.

CHAPTER 3 – RELATED WORK An outline of related researches in the field of FOSS, the development processes, case studies and software engineering methods are presented with appropriate references.

CHAPTER 4 – METHODOLOGY The used methodology and explanations of visualized project data will be provided in this chapter. This includes a general explanation of methods and the presentation of collected data.

CHAPTER 5 – DEVELOPMENT PROCESS ANALYSIS A deeper look is made at several FOSS projects, coming up with an analysis catalogue and applying that on further projects. This analysis will be the center piece on which further chapters build on.

CHAPTER 6 – COMPARISON OF DEVELOPMENT PROCESSES This chapter examines the previous made analysis and compares the findings by working through the established catalogue and analyses.

CHAPTER 7 – DISCUSSION The findings will be evaluated, analyzed and compared with traditional software engineering methods, related findings by other researchers and previously explained methods.

CHAPTER 8 – CONCLUSION The thesis finally concludes with a summary and draws together the main findings of the study along with possible future directions.

2 THEORETICAL BACKGROUND

In order to analyze and compare development workflows, a series of scientific research methodologies were used. This chapter gives an introduction to these methods and tries to vindicate their usage.

2.1 GROUNDED THEORY

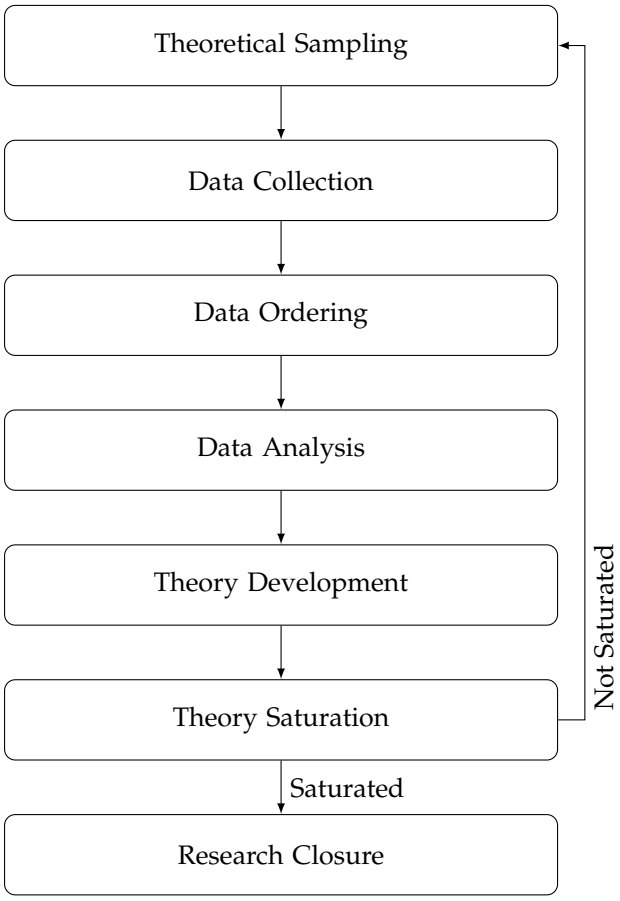


Figure 2.1: Process of theory building using the Grounded Theory as research method according to Strauss and Corbin [Pan96].

The *Grounded Theory* approach is a systematic research methodology first published by Glaser and Strauss [GS67] in 1967. It first was applied in social sciences gathering data and discovering a theory after an analysis of the rallied data. Grounded Theory is mostly used in qualitative research and operates quite different from traditional research methods. It always starts with the collection of data on the researched subject. Thereafter the collected items are marked with

codes which later can be grouped into different categories. They are the base of an emerging theory which can be established out of the gathered data and its analysis.

However Glaser and Strauss have established different opinions on the Grounded Theory approach [Hea04]. This has lead to different appendages, which either favor the Glaser or the Strauss understanding on the application of the Grounded Theory. The most important difference seems to be the technique used to code the samples into categories. Glaser favors a systematical and well defined approach for the coding process while Strauss prefers a more dynamic approach coding the elements as soon as they become visible. The second approach was described by Strauss and Corbin [SC90] in 1990.

Both methods have their benefits and downsides, however in this analysis the Strauss and Corbin approach was chosen as a better fitting research method. Due to the quite dynamic development approach and nescience about the different development processes, a Grounded Theory approach was reviewed as a good scientific research method to gather all the required data and use it as a base research method.

2.2 QUALITATIVE CONTENT ANALYSIS

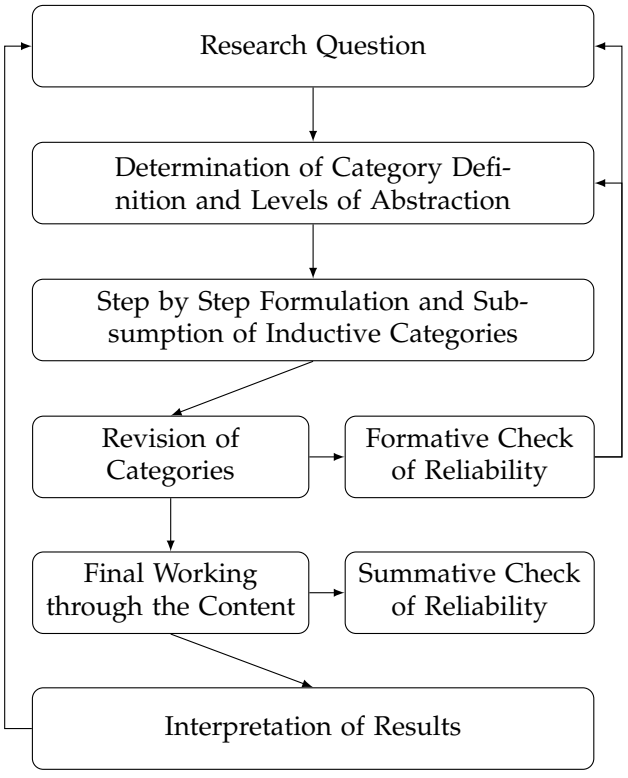


Figure 2.2: Step model of inductive category development in Qualitative Content Analysis according to Mayring [Mayoo; May83].

Another quite useful research method is the *Qualitative Content Analysis* first published by Mayring [May83]. It is related to the previous mentioned Grounded Theory however differs from its theoretical approach. The main idea is to apply methods of Qualitative Content Analysis to the research question and transfer them to a step by step interpretation which can be used for quantitative content analysis [Mayoo]. This leads to a framework one can use to provide an empirical, yet methodologically controlled analysis which follows a previously established content analysis canon.

The centerpiece of this method is the finding and establishment of categories in which the analyzed data can be inserted. It also considers ways to carefully check and review the established categories. This is also known as *feedback loops*. The establishment of categories can be done either in an inductive or deductive way.

For this analysis it makes sense to use an inductive category development, since the research method begins with the research question and the ensuing determination of category definition. This defines what aspects will be taken into account during the analysis. Following this analysis further, possible categories may appear. Those categories can then be revised and checked if they are valid categories for the analysis. Furthermore they can be split up or be set as subcategories of previously found categories. This step will be iterated through the research data until a final interpretation and analysis can be established.

The described Qualitative Content Analysis approach supplements the Grounded Theory in a quite useful way, especially for comparing different development models.

2.3 APPLICATION IN COMPUTER SCIENCE

Qualitative research methods such as the above have their origin in social sciences. However they also have their right of existence in computer science as they provide excellent research methods and tools to examine existing phenomena and domains besides others. As such they are especially useful in areas like software engineering, computer science education or computer science research methods. The following list stands for an array of examples from different areas of the computer science research field.

Hazzan et al. [Haz+06] for example discuss the use and plausibility of qualitative research methods in computer science education. Similarly Meerbaum-Salant, Armoni and Ben-Ari [MSABA10] explore different ways on how to learn computer science concepts with qualitative analysis methods.

Armstrong [Arm06] uses qualitative research methods to discuss the usage of object oriented programming vocabulary in different papers and books. The Grounded Theory is used by Perry, Porter and

Votta [PPV00] to outline the strengths and weaknesses of empirical research in software engineering. It is also used by Sarker, Lau and Sahay [SLS01] for building and developing a process model of collaboration in virtual teams.

Finally also Bainbridge, Cunningham and Downie [BCD03] use the Grounded Theory to analyze music queries. Kaplan and Maxwell [KM94] use qualitative research methods for evaluating computer information systems.

This collective research effort shows the growing interest in using qualitative research methods, especially the Grounded Theory, in computer science. However it shall be noted, that qualitative analysis might not fit every research matter and is often better suited for meta or process analysis.

2.4 SOFTWARE ENGINEERING COMPARISON MODELS

Research in software engineering has brought up many different software engineering models which are suitable for different types of environments. For a later comparison the most renowned and appositely models will be presented in the following.

2.4.1 *Traditional Software Engineering Models*

Many different traditional software engineering models exist in the software engineering research field. The in the following described *Waterfall* and *Spiral model* will stand as an example and be used for the comparison later on.

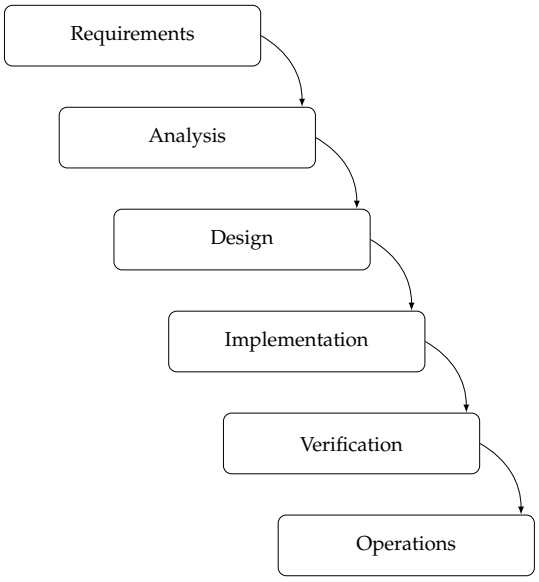


Figure 2.3: The original Waterfall software engineering model with its implementation steps according to Royce [Roy70].

The first formal description of the Waterfall model was published by Royce [Roy70] in 1970. It is interesting to note that Royce did not use the term *Waterfall* to describe his model. He described it as a model which is not appropriate for software engineering. However it is still very popular and became widely known within the software engineering field.

The model follows a sequential process from top to bottom which is the reason of its name. The original sequential phases are from top to bottom System Requirements, Software Requirements, Analysis, Program Design, Coding, Testing and Operations. Furthermore several modified Waterfall models exist, which change the single sequences and domains.

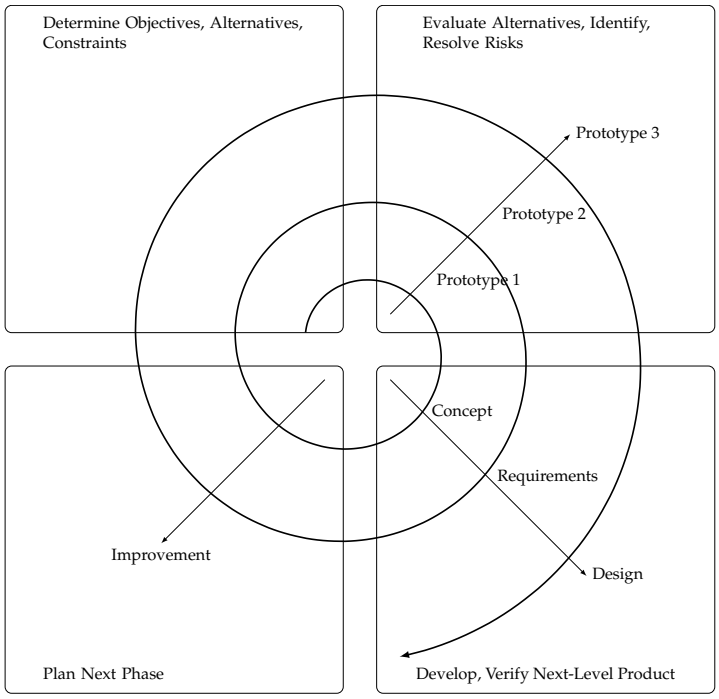


Figure 2.4: The Spiral software engineering model with its typical four step approach according to Boehm [Boe88].

Basili and Turner [BT75] wrote the first description of iterative enhancements inside a software engineering life cycle. In 1988, Boehm [Boe88] created the Spiral model as a response to the Waterfall model. The Spiral model has also been described as an evolutionary or iterative development model. As the name suggest it is devised as a Spiral including steps such as risk management, planning or determination of objectives. It therefore combines the systematic approach of the Waterfall model with an evolutionary development approach which allows to incrementally enhance the product.

A typical cycle of the Spiral always starts with the specification of goals and constraints of the upcoming cycle. It then proceeds to risk management, which tries to identify risks or to provide alternatives.

The defined goals are developed and tested as a next step. Lastly the next cycle is planned.

2.4.2 *Agile Software Engineering Models*

Agile software engineering models base, in comparison to traditional software engineering models, on incremental and iterative development which are known to be quick and flexible to new requirements or changes [Bec99]. The *Extreme Programming* and *Scrum* development models stand as an example for this area.

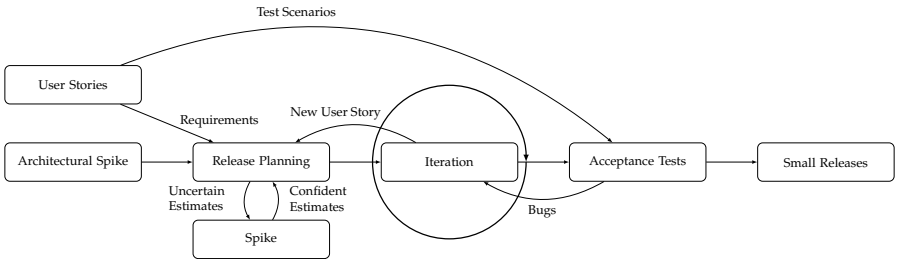


Figure 2.5: The Extreme Programming software engineering model according to Beck and Andres [BA99].

Extreme Programming (XP) is a software development model first published by Beck [BA99] in 1999. Beck set his goal to improve existing software engineering models by improving the overall software quality of a product and the responsiveness of changes required by customers or other incorporated parties. The main differences to existing models are frequent releases and generally short development cycles. It also includes other paradigms such as unit tests, flat management, simplicity, awareness of user requirement changes and frequent communication with the customer. It has to be noted, that Beck does not promote Extreme Programming as a finished concept, but as a process, that existing teams can adapt to [Bec99].

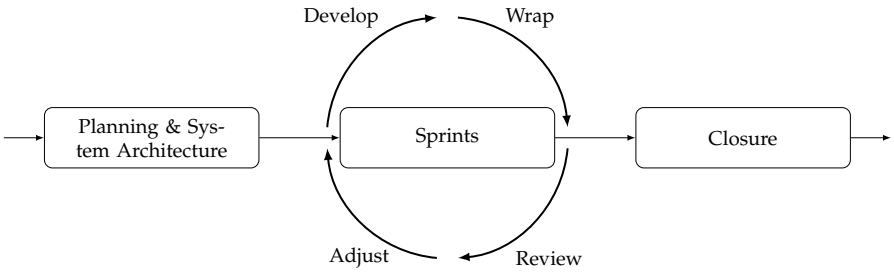


Figure 2.6: The Scrum software engineering model shown as methodology according to Schwaber [Sch95].

The Scrum software engineering model was first referred by DeGrace and Stahl [DS90] in 1990. However, not until 1995 the Scrum development model was presented formally by Sutherland [Sut95]

and Schwaber [Sch95]. Like the Extreme Programming development model it is an incremental and iterative approach. The Scrum development process usually contains three major roles which ensure the process model is followed, represent the customer and the development team. The basic idea behind Scrum are so called *sprints*. These sprints are planned before they are executed and represent one development cycle in which parts of the project will be developed. They usually last between a week and one month. Another important part of the model is the backlog that keeps track of which features are to be done in which priority. These tools ensure the frugalness of new requirements by customers or other constraints.

3 RELATED WORK

Although FOSS is a quite new research subject, many different findings exist in this domain. The most relevant to this analysis will be presented in this chapter.

3.1 PROJECT STRUCTURE

Certainly one of the most relevant and most known studies of FOSS development processes is "The Cathedral and the Bazaar" by Raymond [Ray98]. It is an essay, based on the author's observation which contrasts two software engineering models found in FOSS development. The *Cathedral* model which is planned and developed by a small group of architects and the *Bazaar* model in which the project gets developed in a more chaotic way by a number of developers with no real leader.

The research by Capiluppi and Michlmayr [CM07] draws on "The Cathedral and the Bazaar" and the authors suggest that the Cathedral and Bazaar model are not mutually exclusive, much more they offer the assertion that especially bigger FOSS projects go through both models starting with a Cathedral like development model and then when becoming larger adopting the Bazaar model.

Godfrey and Tu [GT00] on the other hand try to compare FOSS development processes with industrial and traditional management techniques. Additionally they claim that especially big FOSS projects have the ability to still grow at least linearly.

Kim [Kim03] takes a more general approach and tries to describe the FOSS landscape as a whole, the demographics of developers and the structure of FOSS projects. The conclusion is that in most FOSS projects the development is actually led by a small group or single developers which dissents a Bazaar like model. Interesting however is the open approach on information sharing and easily accessible collaboration.

Ogawa et al. [Oga+07] visualized the communication in several established FOSS projects and came to the conclusion that a small group of people were active in most of the discussion about development or future plans.

Also Krishnamurthy [Kri02] makes a strong case for a different development model. In a case study including 100 mature projects the finding was that most projects were indeed lead by small groups of people. Furthermore a Bazaar like discussion was often not happening at all. However the project age seemed to correlate with the

number of developers. In addition the number of project leaders decreased relatively to the size. Similar findings are proposed by Crowston and Howison [CH05] who had analyzed over 120 projects. They suggest however that bigger projects tend to decentralize structures and projects do vary quite drastically in terms of communication structure.

Based on a survey of over 2700 FOSS developers Ghosh [Gho05] assumes a classification of FOSS projects which range from a hierarchical, connected structure to a flat non-connected structure. A similar study which is limited to the Debian project was conducted by Sadowski, Sadowski-Rasters and Duysters [SSRD08].

Finally however the research paper by Conway [Con68] contains an insight that the organization of a software system is similar to the group which designed and implemented the system.

3.2 MOTIVATION

Many research papers exist about motivation of developers or contributers to FOSS projects. Lakhani and Hippel [LH02] for example examined why people do provide free support to other developers or users. They claim that most people do offer support because it returns direct learning benefits. Lerner and Tirole [LT00] focus more on developers.

Also Grazzini [Gra09] questioned why developers would work for free and came to the conclusion that a complex interaction between several technological, social and economic factors provide reasons for a developer's motivation.

Lakhani and Wolf [LW03] implemented a web based survey analyzing the answers of over 600 developers and over 287 projects. They consider external motivational factors as implausible and propose enjoyment-based intrinsic motivation as the main motivation for professionals but also for volunteers. A similar study, but limited to the Linux kernel was done by Hertel, Niedner and Herrmann [HNH03]. The findings were similar, however analyzed from a psychological point of view.

3.3 SOFTWARE ENGINEERING

A descriptive analysis of FOSS development is offered by Roets, Minnaar and Wright [RMW07]. The authors claim that no single software development process exists. Yet they derive a software development cycle based on different established software engineering processes.

Warsta and Abrahamsson [WA03] discuss whether agile methods and FOSS development methods are similar. They have come to the conclusion that there are similarities but also considerable distinctions. They suggest however that both methods can learn from an-

other. A similar finding is provided by Koch [Koc04] who names the biggest difference the co-location in agile methods, which of course is not available in most FOSS projects.

Spinellis and Szyperski [SS04] make a strong case that FOSS affects traditional software development since many FOSS projects do actually share pieces of code or use other projects for their development. A concrete analysis of software engineering processes in the GNOME project is offered by German [Ger03].

Taking a look on the evolution of software engineering processes Scacchi [Sca06] provides a study claiming that development processes evolve together with the community of a project and that their development processes have great influence on companies who were used to traditional software engineering processes.

3.4 CASE STUDIES

In addition to the research studies, many case studies are relevant to this analysis and focus on several FOSS projects. Mockus, Fielding and Herbsleb [MFH02] analyze Apache and Mozilla and compare them with several commercial projects. Dinh-Trong and Bieman [DTB04] provide a case study on the FreeBSD project with a final comparison with the Apache project.

Crowston et al. [Cro+04] analyzed several FOSS projects and offered an analysis of several project success measures. Another case study is provided by Magnusson [Mag10] who considered only the PHP project. Similarly a case study of the GNOME project is provided by Koch and Schneider [KS02]. Furthermore the Plone project was analyzed in depth by Aspeli [Asp05].

Johnson [Joh01] provides a quite exhaustive process model of different FOSS projects. He comes to the conclusion that most FOSS projects follow an adaptive life cycle. Almost all analyzed projects have established a flexible management model based on leadership, collaboration and accountability.

A theoretical model of the structure of FOSS projects is offered by Crowston et al. [Cro+05]. The proposed model focuses on software development, distributed work and the structure of distributed teams.

4 METHODOLOGY

Based on the scientific research methods explained in chapter 2, this chapter will give an explanation of the used methodology, research choices as well as the approach to the different projects.

4.1 PROJECT SELECTION

Since there is a huge number of FOSS projects to choose from, this study focuses on well established projects with a quite long history compared to others. To achieve a good selection of such projects several criteria were established to ease the assortment. These criteria allow a good selection and furthermore they should give a reasonable sample within this analysis can produce valid results.

4.1.1 *Category*

There is a vast number of FOSS projects available. As of course not all projects have the same goals in mind and provide similar software, it makes sense to differentiate the projects by their goals and the type of the resulting software. Examples are programming languages, desktop interfaces and other. Such a breakdown enables a direct comparison of projects with similar goals as well as with other, not related projects.

4.1.2 *Popularity*

In order to not choose small or not relatively unknown projects, the popularity and usage of a project was taken into account. While it is very difficult to represent this factor with absolute numbers, the number of active developers, approximations of installations and citations in websites or magazines give a good quantitative representation.

4.1.3 *Project Age*

It seems that especially FOSS projects take some time to identify their development workflow and structure their project accordingly. Furthermore quite new projects often do not have such a structured development workflow and it is seldom known if they still will be developed in the near future. As such a minimum project age was set to ten years. To exclude older projects, which are only maintained and

do not follow a typical FOSS workflow the maximum project age was set to 25 years.

4.1.4 *Activity*

The activity of a project is a very important measure to include only actively developed projects which still fall into the above mentioned time frame. The activity of a project however is difficult to measure, but one can look at certain numbers published by most of the projects which allow a good insight of how active the project is without rating it quantitatively.

RELEASES Regular or a high number of releases are a sign of a high activity of FOSS projects. As such this represents an important criteria for the selection.

DOWNLOADS Some projects publish the number of accesses to their code repositories or number of downloads of their releases. A big number of course shows a high popularity and also a high activity within the project.

NUMBER OF COMMITS There is a correlation between the number of commits to a project's repository and its activity. As such a large number of commits also shows a high activity.

4.1.5 *Community*

The people behind a project are the driving force, directly lead and develop a project. As such it makes sense to provide some criteria to choose projects with a considerable number of developers.

COMMUNICATION In order to be able to examine development workflows in a project, the communication methods have to be openly accessible. Furthermore a project should have a good level of communication over diverse methods such as mailing lists, forums or chat systems.

DEVELOPERS The number of developers of a project is a direct indicator of the popularity of a project, its size and activity. As such a large number of developers is wanted for this analysis, however it is not always possible to get an accurate number.

CONFERENCES Meetings such as conferences, hackfests or other are an indicator of an active and vital community. They represent an important meeting point in most projects and as such are a good criteria for the activity of projects.

FOUNDATIONS Bigger FOSS projects mostly have a foundation as backer, which makes sure that the project stays independent from companies or other. They usually appear after a few years of development or a growing size of the project.

ONGOING PROJECTS A lot of bigger FOSS projects hold or join projects like Google's Summer of Code program in which students are invited to join the project for a few months. Such programs almost always lead to a larger community which is able to host such events.

4.2 FINAL SELECTION

With the above explained criteria catalogue, ten FOSS projects were chosen and analyzed. This leads to the following list of projects.

PROJECT	ORIGIN	CATEGORY
Debian	1993	Operating System
Drupal	2001	Content Management System
Fedora	2002	Operating System
GNOME	1997	Desktop Environment
KDE	1996	Desktop Environment
MySQL/MariaDB	1997	Database Management System
PHP	1994	Programming Language
Plone	1999	Content Management System
PostgreSQL	1986	Database Management System
Python	1989	Programming Language

Table 4.1: List of analyzed FOSS projects.

A deliberate choice was made to only analyze the core parts of the projects, even if they are built with a modular approach or consist of several other parts. This choice was made to allow an easier comparison between the projects without any clutter they might bring in.

4.3 VISUALIZATION OF PROJECT DATA

For this analysis not only the publicly available project information was analyzed, but also the code repositories of the projects. The reason for this is to underly the made conclusions and to check the analysis for validity. For FOSS projects all code repositories are publicly available and often range back to the beginning of the project. In some cases however data from the beginning is not available, be it because of the lack of versioning control systems at that time or

migrations of version control systems. In other cases the code was split into several different code repositories, which made it harder to combine the given data. With those two points in mind, all software repositories were downloaded and prepared for automatic analysis. This produced several graphs which will be explained below.

4.3.1 *Commits by Author*

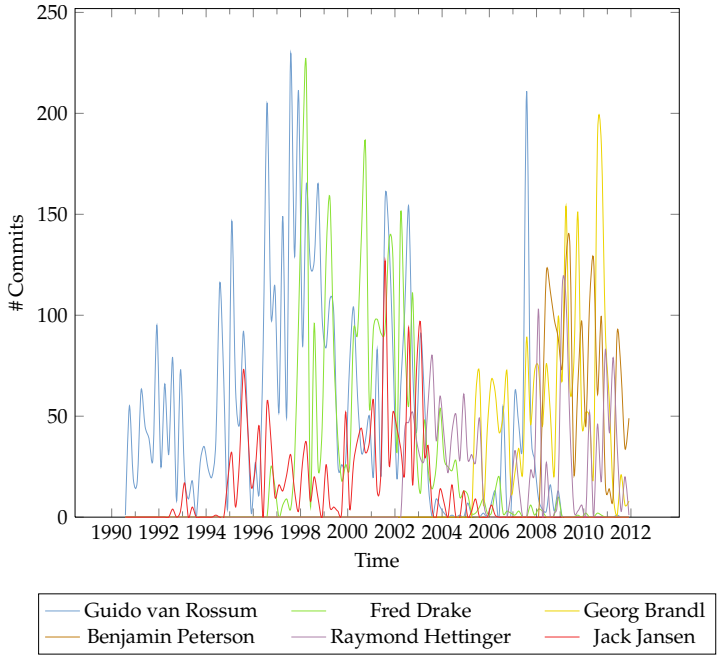


Figure 4.1: A graph displaying the six most active developers with their monthly commit number with data from the Python project.

The above graph shows the six most active developers of a project with their over time commits per month. It quickly allows to see the involvement of certain people in the project and their activity on the development side of the project. In this case for example the Python project leader Guido van Rossum diminished his activity from 2004 on in comparison to other developers. This could mean two things, first a person could have left a project or, which is the case in the Python project, the person is busy in other parts of the project.

For the generation of this graph all related software repositories were analyzed in relation to each developer with the most commits. The first six were then chosen and their monthly commit count was analyzed from the projects' inception (if available) until the end of the analysis. The monthly number was then plotted and a curve was interpolated between these points. To smooth the curve and to allow an easier understanding of the graph each second month was left out and interpolated over the other points.

4.3.2 *Commits by Year*

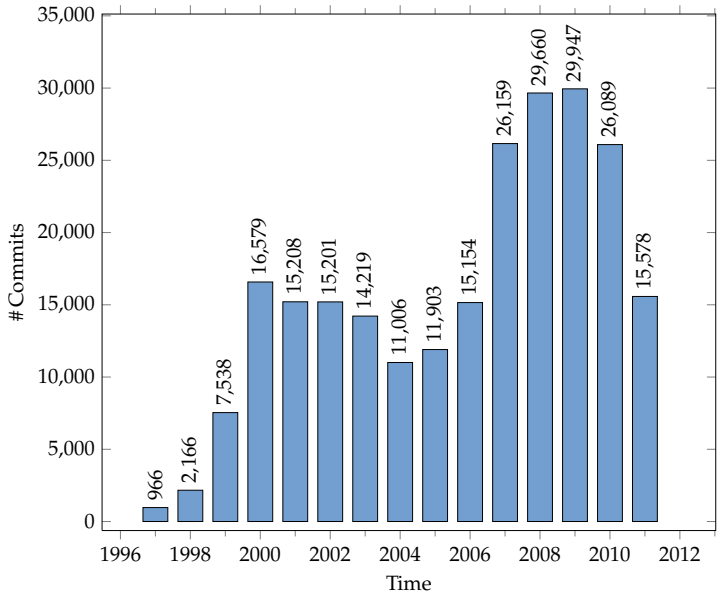


Figure 4.2: A graph displaying the number of commits per year with data from the KDE project.

The above graph shows the number of total commits per year. Most projects have a steadily increasing number over the years. In some cases like the above there is a quite large leap. In the above case this can be explained with the development of a new major KDE release.

The graph was generated counting all commits per year from inception till the end of the analysis. It is interesting to note that this graph often matches up with other graphs in the same project, such as the commits by month or the authors by month graphs.

4.3.3 *Time Based View*

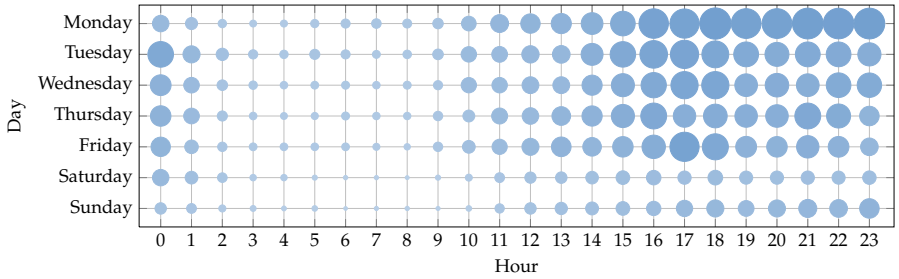


Figure 4.3: A punchcard graph showing when commits usually occur with data from the GNOME project.

This graph shows all commits from inception till the end of the analysis in form of a circle with the related day and time combina-

tion. A bigger circle represents more commits at the given day/time combinations. In the below example the most busy day was Monday between 4 pm and 12 pm. This can be easily explained with the fact that the GNOME project usually does new releases on Monday. Furthermore such a graph shows the developers usual working hours. This example also shows that many developers tend to have a main job in a company, as usual office hours match up with the day and time combinations. This is enforced by a quite low number of commits on Saturday and Sunday in comparison to workdays.

The graph was generated counting the commits depending on their day and time. Of course local time was converted to Coordinated Universal Time (UTC) to guarantee comparable results. The circles were then drawn on the graph using different sizes and transparency depending on the number of commits on a single day and time combination. The size and transparency is a relative value ranging from the day and time combination with the least number of commits to the day and time combination with most commits. All other circles were then drawn using a value between this range.

4.3.4 *Commits by Month*

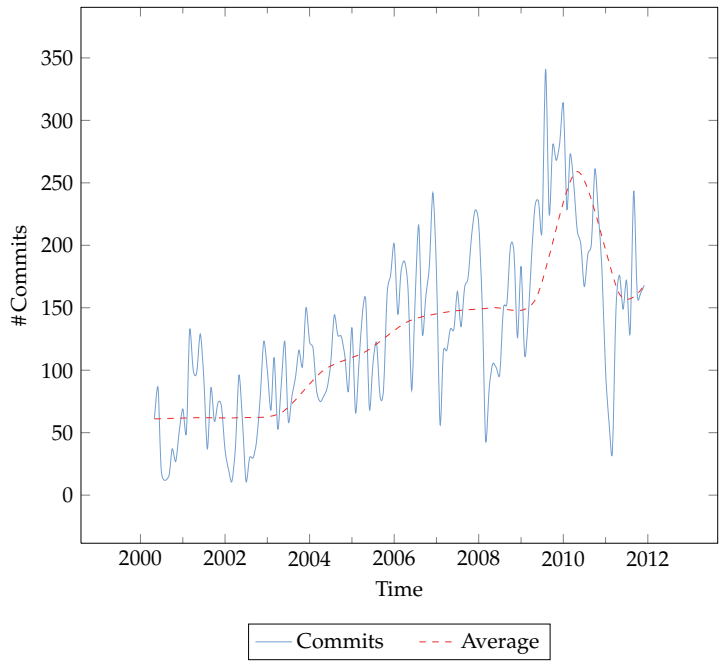


Figure 4.4: A graph showing the number of commits per month with data from the Drupal project.

The above graph shows the commits per month correlation from inception till the end of this analysis. It is quite similar to the yearly overview mentioned before, however gives a better insight into the monthly development. An increase of commits always shows a cer-

tain degree of interest of people for the project. It can of course mean that more people are willing to contribute to a certain project, but also that the existing developers are producing more code. This is especially interesting when comparing it with the following graph. In the above example there is a peak between 2008 and 2009 which can be explained with the upcoming Drupal 7 release development at that time.

The graph was generated counting the commits from inception till the end of this analysis and plotting that number for each month. The blue line then was interpolated between those points. The red line on the other hand stands for the average value over a year and is useful to identify the project's direction.

4.3.5 *Authors by Month*

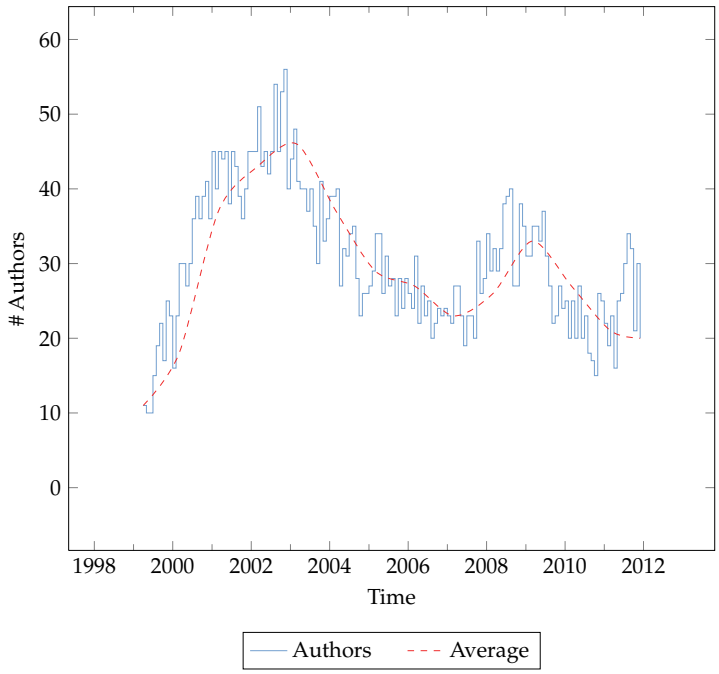


Figure 4.5: A graph showing the number of distinct authors per month with data from the PHP project.

The above graph shows the number of distinct authors per month from inception till the end of analysis. It gives a good impression on how many people were responsible for the amount of work done in a certain time period, which can be seen in the previous graphs. The above example shows a decrease of authors meaning either that developers left the project or that developers reduced their time working on the project.

The graph was generated counting all distinct authors per month from inception till the end of analysis. As this is always in whole numbers, the graph was generated using a so called constant plot. The red

line on the other hand is an average value over one year which is interpolated between those points. In case of authors with different email addresses or slightly different names in the single commits, they were combined if it was the same person.

5 DEVELOPMENT PROCESS ANALYSIS

In the following chapter the chosen FOSS projects will be analyzed in depth. After a thorough analysis of each project, a project analysis catalogue will be proposed. This first part of this chapter makes heavy use of the Grounded Theory research method presented earlier. The second part of this chapter will then use the proposed catalogue to analyze further projects. This is a good use case for Mayring's research method for Qualitative Content Analysis described in chapter 2.

5.1 DRUPAL PROJECT ANALYSIS

Drupal is a Content Management System (CMS) and a Content Management Framework (CMF). It is written in the PHP programming language and licensed under the GNU General Public License (GNU GPL). Drupal is used for a wide range of capabilities such as blogs, company and political websites, communities and is directed to be a widely spread CMS [Dru#5]. Drupal is also known as *Drupal Core* inside the Community and contains the base functionality of the CMS. However it can be extended through numerous modules. In the following though only Drupal Core will be analyzed.

5.1.1 *History*

The first version of Drupal was released in 2001 by Dries Buytaert [Dru#4]. Originally the software provided a black board like directory. Quickly however a fully-fledged CMS emerged out of the initial version. The name originates from the dutch word *druppel* which translated means *drop*. Especially in the last years it experienced a wide distribution and is used for at least 1.7 % of all websites worldwide and holds a market share of 6.2 % [Bui; W3T#1].

5.1.2 *Community*

The Drupal project has a large user and developer community around the world. According to own statements, over 250.000 users were registered on `drupal.org` in August 2011. About 1200 of those were additionally listed as developers, whereas the number could be a bit higher [Buy11].

Twice a year the community holds a conference under the name *Drupal Conference* or *DrupalCon*. To alleviate the journey of the atten-

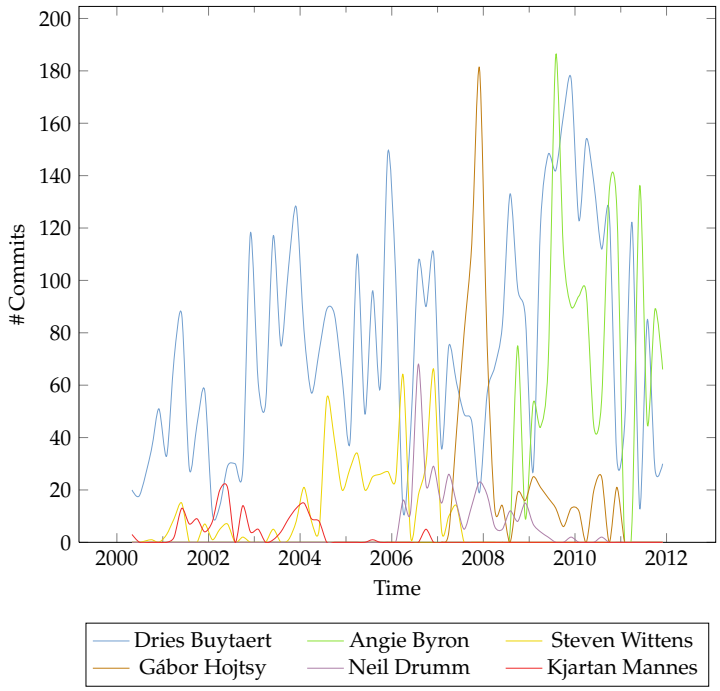


Figure 5.1: Monthly activity of the most active Drupal Core developers. It clearly shows the involvement of the founder Dries Buytaert. In recent years however the roles of Gábor Hojtsy and Angie Byron became more important.

dees the conference takes place alternatively in Europe and North America.

The communication possibilities sprawl from forums on `drupal.org`, mailing lists and discussion groups on `groups.drupal.org`. Additionally a wide range of Internet Relay Chat (IRC) channels for different aspects of the Drupal project are being used.

Next to the last-mentioned classification of users and developers, people can – provided that they are working on Drupal Core – be classified into one of the following categories [Dru#1].

CORE CONTRIBUTOR All developers who bring in code or documentation for Drupal Core are known as core contributors. All modification proposals are checked by another core committer in a review process and are then either accepted or dropped.

MAINTAINER While maintainers often aren't involved in the decision making progress, they have the liability of small portions of Drupal Core. Most of them are particular core modules or technical domains such as JavaScript. The assignment happens over Dries Buytaert, whom interested developers can approach or will be invited to join.

VENUE	DATE	ATTENDEES
Munich, Germany	August 2012	N/A
Denver, USA	March 2012	N/A
London, England	August 2011	1751
Chicago, USA	March 2011	3000
Kopenhagen, Denmark	August 2010	1200
San Francisco, USA	April 2010	3000
Paris, France	September 2009	850
Washington D.C., USA	March 2009	1400
Szeged, Hungary	August 2008	500
Boston, USA	March 2008	850
Barcelona, Spain	September 2007	450
Sunnyvale, USA	March 2007	~300
Brussels, Belgium	September 2006	150
Vancouver, Canada	February 2006	~150
Amsterdam, Netherlands	October 2005	~100
Portland, USA	August 2005	100
Antwerpen, Belgium	February 2005	~50

Table 5.1: Previous and planned DrupalCon events [Wal11].

CORE COMMITTER Only few people have write access to the Drupal Core repository. They are known as core committer and look through code changes and propositions and decide about patch acceptances. At this juncture there is a distinction between permanent core committers and branch maintainers.

PERMANENT CORE COMMITTER Dries Buytaert is currently the only permanent core committer.

BRANCH MAINTAINER Next to Dries Buytaert they are in charge of maintaining a particular Drupal version.

- Gerhard Killesreiter for Drupal 4.7.x.
- Neil Drumm for Drupal 5.x.
- Gábor Hojtsy for Drupal 6.x.
- Angie Byron for Drupal 7.x.

FOUNDER AND LEAD DEVELOPER Dries Buytaert, the founder of the Drupal project, holds the role of the project leader and therefore decides the project's direction. Furthermore he is the main decision maker. Though in some cases he also gives away his decision control to a trusted person.

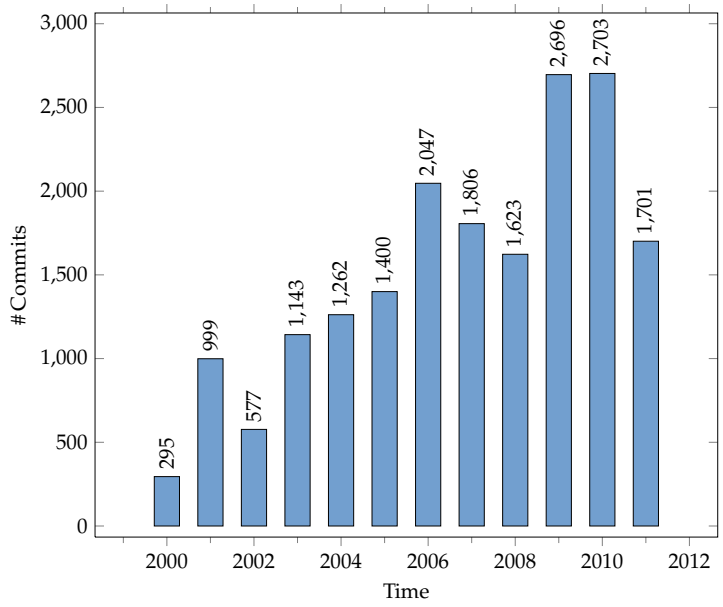


Figure 5.2: Yearly overview of commits to Drupal Core. The peak in 2009 and 2010 is most likely related to the development of Drupal 7 which was released in early 2011.

5.1.3 *Release Process*

For releases, Drupal uses a version naming with two numbers since Drupal 5. The numbers stand for major and minor versions [Dru#6]. New major releases are quite scarce and will get released every few years. They are planned for a long time in advance, often with incompatible changes. Before Drupal 5 a three numbers versioning scheme was used.

While the time periods between single Drupal Core major releases are not bound, the time until a stable release is divided into single phases [Dru#3]. The length of a single phase is not fixated either, but is limited by a range of factors which will be explained in the following.

CODE THAW After each major release a new branch will be created in the Drupal Core repository. In this phase all new features and modifications can be discussed and added. No restrictions apply.

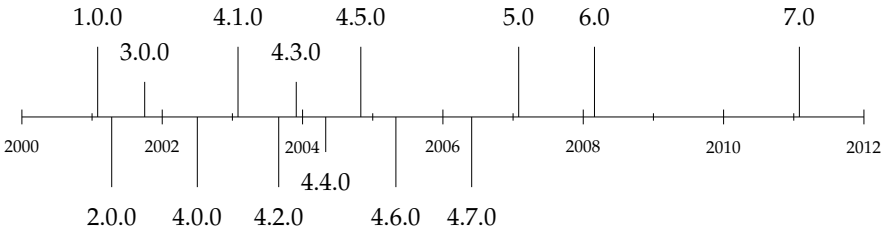


Figure 5.3: Major releases of Drupal.

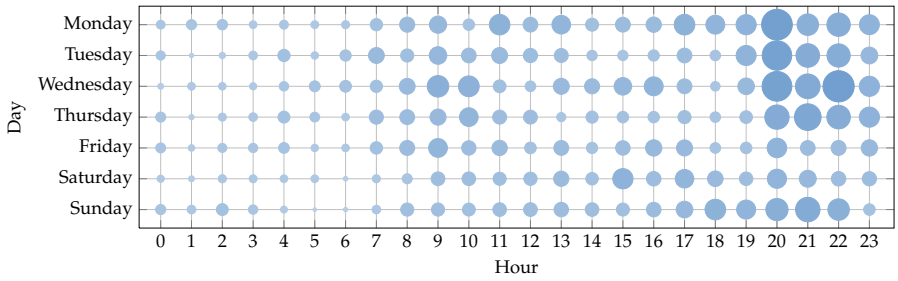


Figure 5.4: Time based view on commits of core contributors. There seems to be a tendency to evenings during the week and Sundays which could be interpreted with a more community driven project.

CODE SLUSH This phase was added to the Drupal 7 release cycle to restrict the time between major releases. Important features will be elaborated and handled with the help of so called initiatives. In this phase no new features except the chosen initiatives are allowed. Though changes will be accepted to fix errors or add missing functionality.

At the end of this phase, the Application Programming Interface (API) freeze will be declared. At this point in time, the API can only be changed when critical errors are found and have to be corrected.

POLISH PHASE In this phase Drupal Core will be polished and the last enhancements on performance, accessibility, documentation and other are brought in. At the end of the phase string freeze and user interface freeze will be put in place.

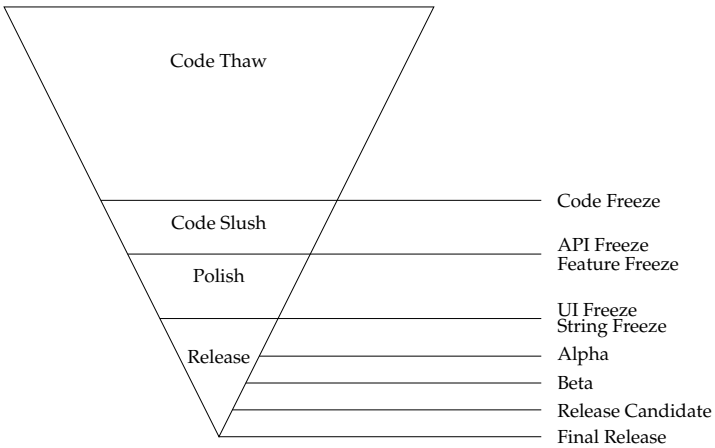


Figure 5.5: Drupal release process phases.

RELEASE PHASE Lastly, several alpha, beta, release candidates and in the long run the stable version will be released. The first release candidate is published once there are no more known critical bugs.

Additional release candidates are released if new critical bugs appear, otherwise the final release is published.

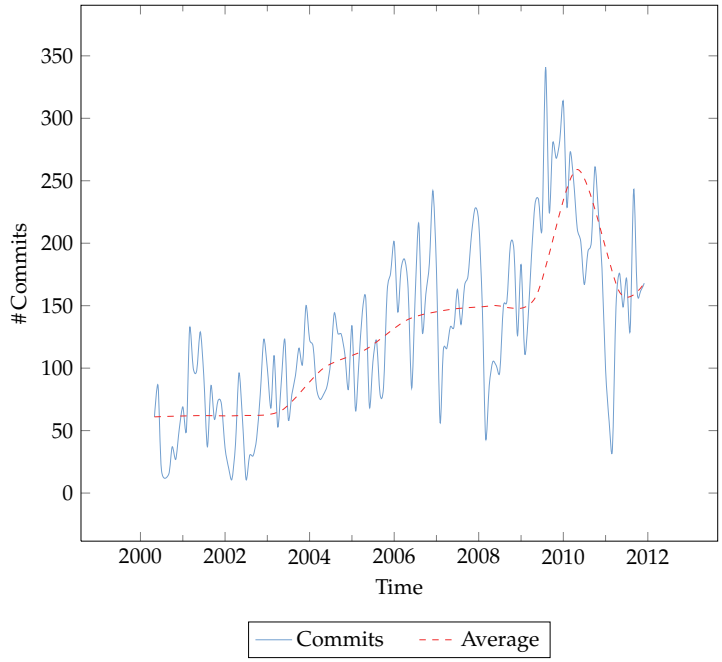


Figure 5.6: Amount of commits per month of core contributors. Again, the peak in 2009 and 2010 seems to match the development of Drupal 7. The subsequent low is then related to the planning phase of Drupal 8.

5.1.4 *Development*

The Drupal project changed its development workflow for the upcoming Drupal 8 release. Until then, the development was mostly driven through the issue queues, which is a bug tracking system on `drupal.org`. While it is still used for communication, bugs and new features, the development process features a number of core initiatives which will stand for a major area of Drupal [Dru#2]. Each initiative has one or two initiative owners who lead the development and coordinate the initiative while working closely with Dries Buytaert.

Initiatives can be created by any Drupal developer, however Dries Buytaert choses which initiative will be followed for the next major release. He also choses the initiative leaders and remains in close contact with them. Furthermore each initiative defines a small set of goals which should be reached by the time of the next major release. For the Drupal 8 release, the following initiatives will be followed:

- Configuration Management
- Web Services Context Core Initiative
- Design 4 Drupal

- Drupal 8 Multi-Lingual Initiative
- HTML5
- Mobile

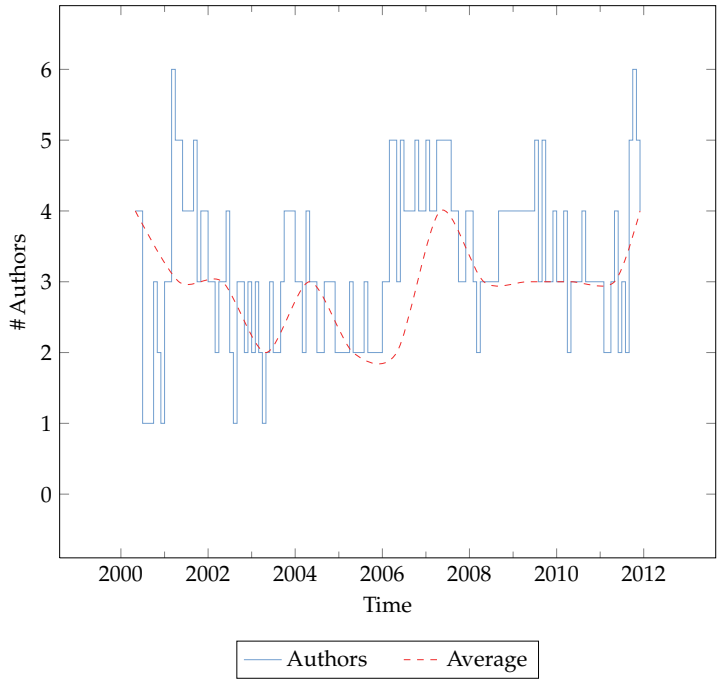


Figure 5.7: Amount of distinct authors of Drupal Core over time. Due to their policy, only certain developers are allowed to commit to the repository. The project recently started adding author names to the commits.

5.2 PLONE PROJECT ANALYSIS

Plone is a CMS built on top of the Zope application server which provides the core functionality layer for the project [Aspo5; Plo#6; Plo#13]. It is mostly written in Python and available for all major platforms. Plone is licensed under the GNU GPL and according to Ohloh, Plone is one of the top 2 % of FOSS projects worldwide [Bla11]. Furthermore it is used by organizations and companies such as NASA, Amnesty International, Nokia and others. It can be used for a wide range of websites such as internal websites, blogs, groupware and other. The project is known for its very good security track record, high usability, extensibility and flexibility. The project is split into Plone Core and Collective. This analysis however will only focus on Plone Core as it is the heart of the project.

5.2.1 *History*

In 1999, Alexander Limi and Alan Runyan started the Plone Project creating a usability layer on top of the application server and CMF Zope [Asp05; Plo#6]. The first version was then released two years later in 2001 [Plo#12]. It was quickly picked up by many people and a community around the Plone project began to emerge. In 2004 Plone 2.0 was released which made Plone more configurable and enhanced the possibility of adding modules to the CMS. At the same time, the Plone foundation was established, which to this day has ownership rights over the Plone project and trademarks. The next major Plone version was released in 2007 under the name Plone 3. The currently used and developed Plone 4 was released in 2010.

5.2.2 *Community*

The Plone project consists of a large user and developer community. While there are about 300 people working on Plone Core, there is a much bigger community working on Plone Collective projects [Asp05; Bla11; Plo#1].

Most of the communication is done through the Plone mailing lists whereas the *plone-developers* mailing list is in place to discuss the development and future of Plone Core. There are also lots of other mailing lists available for almost every aspect concerning the project. However, there are also large forums and IRC channels available where one can give and find support.

The first Plone conference was held in New Orleans in 2003 [Plo#5]. Since then the yearly Plone conference attracted a growing number of attendees every year and was held in Europe as well as in the United States.

The Plone community is very eager to organize so called sprints [Plo#11]. A sprint is a focused development session where developers meet in person and try to enhance a certain part of the Plone project. It normally lasts about three to five days in which one person leads the session and tracks the activities and the development. Many sprints take place worldwide every year.

Due to the large group of Plone developers, the project is split into several teams, where each team is responsible for a specific part of the Plone project such as release management, development, security, infrastructure, usability and more [Plo#7; Plo#9; Plo#8; Plo#2].

CORE DEVELOPER This role is defined to have access to the Plone Core codebase. To become a member of this role one must have established a track record of contributions to the Plone project. This is not limited to code, but can also be usability enhancements or other contributions. It is also highly recommended to attend a conference

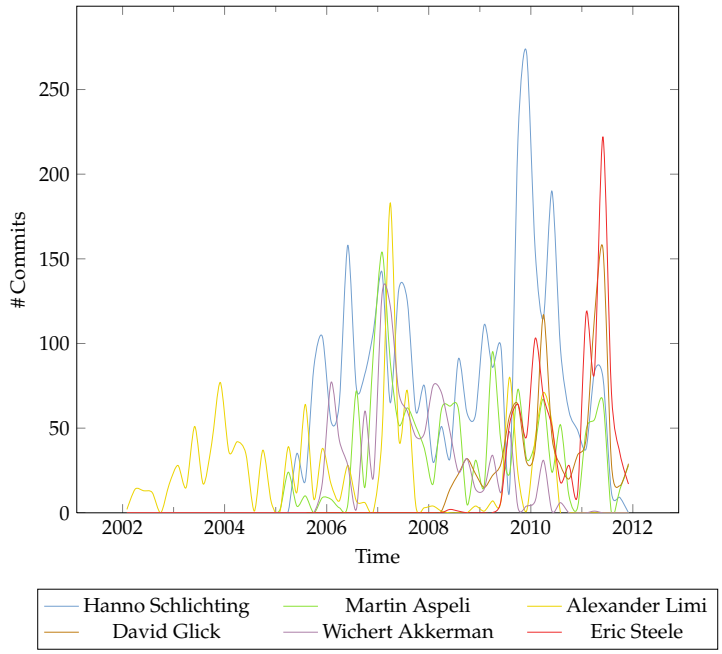


Figure 5.8: Monthly activity of the most active Plone developers. While the founder Alexander Limi was very active in the inception phase, Hanno Schlichting and Eric Steele seem to drive the project much more in present time.

or sprint in order to become acquainted with the other developers. Finally one has to request an account. Informally, Godefroid Chapelle, Hanno Schlichting and Martin Aspeli serve as gatekeepers for this role. Currently this role has 375 members.

FRAMEWORK TEAM This team is responsible for the decision of which code gets into a Plone release. They do this by recommending code for inclusion to the Plone release manager team. They are however not responsible for adding new features or patching bugs, they only drive the Plone Improvement Proposal (PLIP) process. By accepting or rejecting PLIPs they gather information from the community and finally recommend them to the release manager. Furthermore all significant changes must go through the framework team. The team consists of a small group of people and are chosen arbitrarily. Currently the team consists of 16 people.

RELEASE MANAGERS Each major Plone release is done by a release manager who is responsible for continuing releases in the specific series. The release manager gets named by the Plone foundation board and is a paid position [Plo04]. The manager works closely with the framework team to ensure improvement and continuing development of Plone. The current release managers are Alec Mitchell, Eric Steele, Hanno Schlichting, Stefan H. Holek, Wichert Akkerman, Alan Hoey.

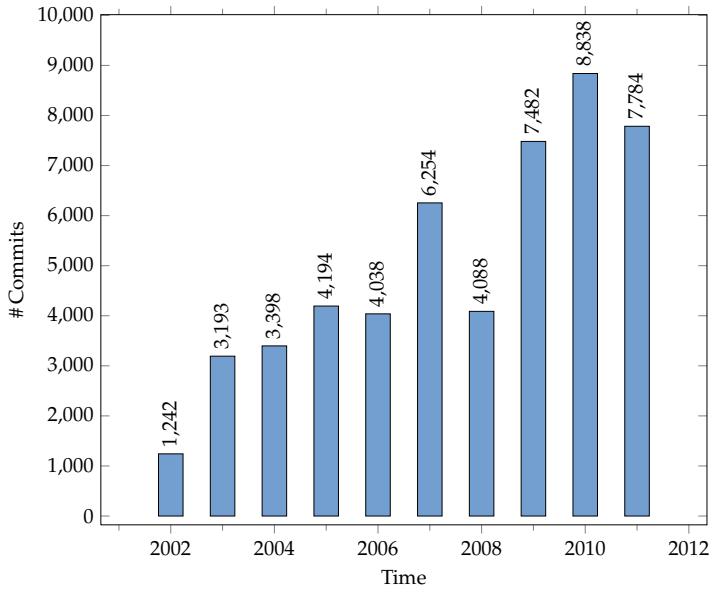


Figure 5.9: Yearly overview of commits to Plone. The steady increase of commits is interesting to note with a boost before the Plone 4 release in late 2010.

PLONE FOUNDERS The Plone founders are Alexander Limi and Alan Runyan. While they do not have any further rights, both still have a powerful voice in the community and help with decisions where the community is not able to take one. In some cases they also take a decision by themselves, overruling the community's proposal.

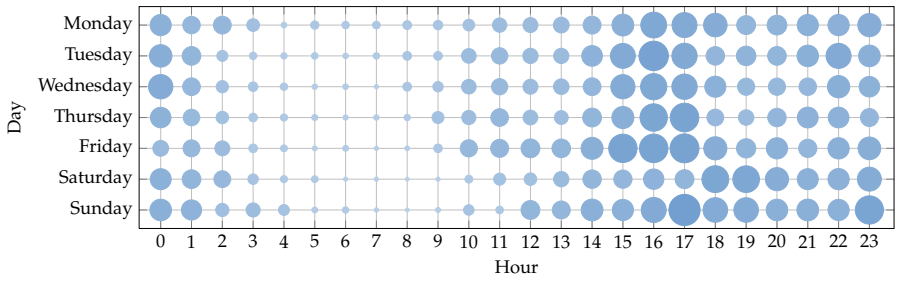


Figure 5.10: Time based view on commits of contributors. The times seem to be equally distributed between office hours and evenings or weekends, which could be interpreted with an equal amount of employed and volunteered commits.

5.2.3 *Release Process*

Since the Plone 4.2 release in 2011 the Plone project changed it's release process in order to implement a fixed release cycle [Plo11]. Now every six months, there will be a new Plone release. Additionally the Plone project sets a feature freeze date two months before the release

at which no new PLIP will get accepted unless it has been completely reviewed by that date.

The Plone project uses a major, minor and micro versioning scheme [Plo#10; Plo#1]. Major releases break backwards compatibility but may provide an upgrade path. Minor or feature releases normally provide new features and bug fixes but retain backwards compatibility. Micro releases are only bug fix releases and do not contain new features. Additionally, each major and minor release has several preceding releases. Alpha releases are basically snapshots of the current development process. Once a beta release gets published, no new feature is allowed to enter the codebase. Only bug fixes and non-invasive changes are allowed. A release candidate is published afterwards and only if problems are found, further release candidates follow. If no show stopper bug comes up, the last release candidate is considered to be the final version and gets published on the previously set release date. After that date only bug fix releases are allowed for this series. The development starts with the next major or minor release.

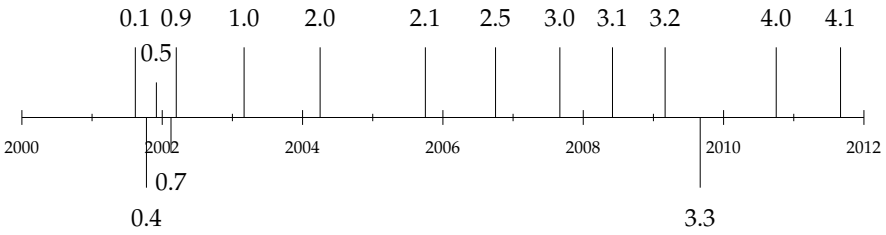


Figure 5.11: Major and feature releases of Plone.

5.2.4 *Development*

Most of the development of the Plone project is handled through the mailing list and bug trackers of the project [Plo#2; Plo#1]. Minor patches and contributions often get committed directly to the repository while bigger changes and features have to go through the PLIP process [Plo#4; Plo#1; Plo#3]. This process is based on the Python Enhancement Proposal process. New features and big changes have to be written down in a PLIP which contains a description of the feature or problem, a solution to it and a working implementation. A PLIP is always owned by at least one person who is fully responsible for it.

A PLIP mostly derives from a discussion on the *plone-developers* mailing list. The community will give feedback on the idea and possible solutions. If the feedback can be considered positive, the creator of the idea will be asked to write a PLIP. However one can also start directly with the creation of a PLIP and get the community feedback during the upcoming phases.

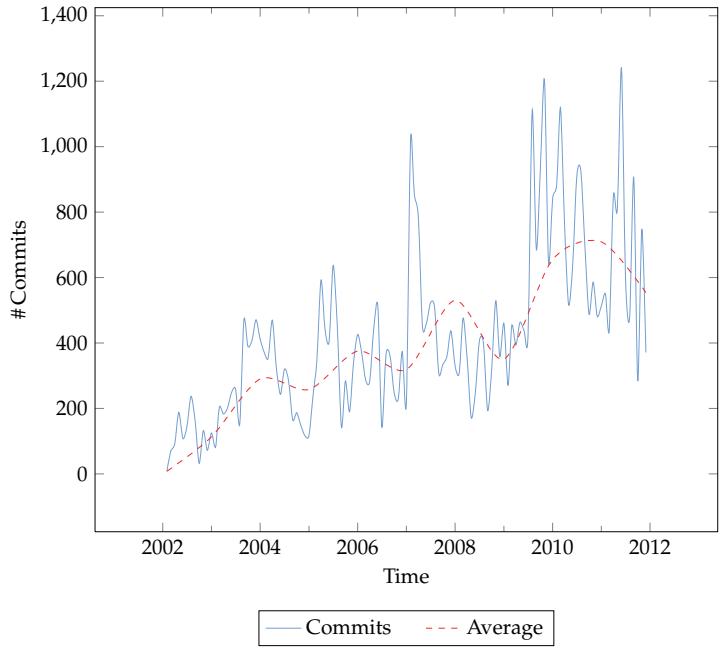


Figure 5.12: Amount of commits per month of core contributors. The two peaks seem to refer to the development of Plone 3 and Plone 4.

The author of the PLIP is always considered as the main contact point for future improvements or discussions. From here the PLIP life cycle begins [Plo#3].

DRAFT A PLIP always starts as draft, where the author summarizes his ideas and tries to come up with a solution. It is not ready for a proposal but the discussion already starts and new solutions or ideas find their way into the PLIP.

PROPOSAL Once the author thinks that the PLIP is finished and will get accepted they can submit it at any time. The framework team will then comment on any newly submitted or updated PLIP. It will be reviewed and if the framework team decides that the PLIP won't get accepted for Plone Core, the author is encouraged to improve the PLIP and resubmit it at any later point in time.

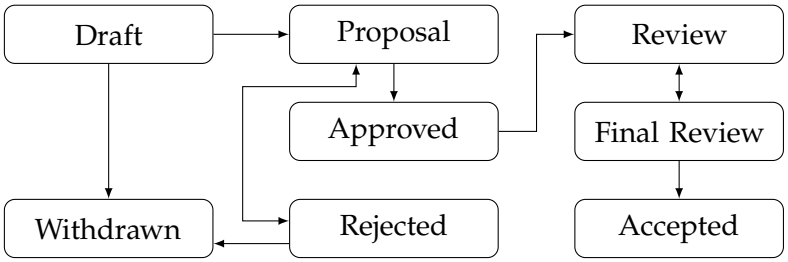


Figure 5.13: Possible paths of the status of Plone Improvement Proposals.

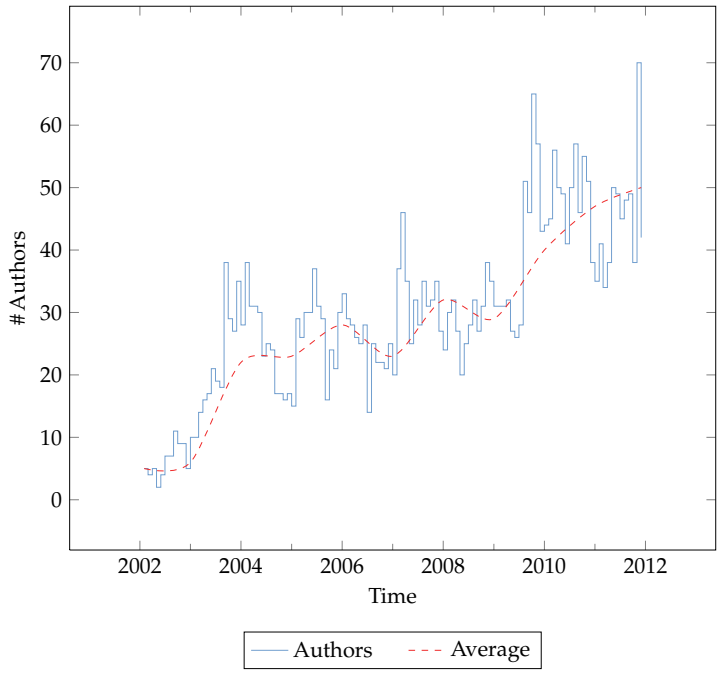


Figure 5.14: Amount of distinct authors of Plone Core over time. The development of Plone 4 obviously did attract new developers.

APPROVED As soon as the PLIP is approved, the author of the PLIP starts with the development of the PLIP. A member of the framework team will get assigned to the PLIP and assists the author with any questions and help they might need.

REJECTED If the PLIP was rejected by the framework team, in most cases the PLIP gets implemented as an amendment and later submitted again as an improved PLIP.

REVIEW Once the development is complete, the author of the PLIP can forward the implementation to the framework team. Two members will then review the proposal along with two additional developers chosen by the author. The reviews have to be in written form and publicly available to a all members of the community. This phase should not take longer than two weeks and all critical bugs have to be fixed before that. If there are too many changes for the available time, the PLIP will be pushed to the next release.

FINAL REVIEW Once all major concerns have been solved, the original reviewers will review the PLIP one more time and confirm that all issues have been resolved and no new issues have been found. If issues have been found, it goes back to the Review state.

ACCEPTED If the PLIP passes the Final Review phase, it can be merged and will be available in the next Plone release.

WITHDRAWN An author can of course always decide to withdraw the PLIP if they think, that the PLIP is no longer needed or obsolete.

5.3 PYTHON PROJECT ANALYSIS

Python is an object oriented and interpreted programming language well known for its simplicity and easy to learn syntax [Pyt#6]. Licensed under the *Python Software License* [Pyt#4] it is available for all major platforms. Python comes with an interpreter and an extensive standard library. It supports many programming paradigms, such as object oriented, imperative or functional programming styles. The reference implementation of the interpreter is called *CPython*, however, there are other interpreters available as well. The subject of this analysis will be CPython.

5.3.1 *History*

The creator of Python, Guido van Rossum, conceived and implemented Python around 1989 at the CWI Institute in the Netherlands [Ven03]. Python was a successor to the ABC programming language, which itself saw contributions from van Rossum in the eighties. The name alludes to the British television series *Monty Python's Flying Circus* van Rossum was watching while creating the programming language. He still is Python's principal author and project leader and therefore the project's main decision maker.

Currently, there are two versions of Python available, both incompatible to each other. Python 2.0 was first released in 2000 and since then evolved to the currently available Python 2.7 version. Python 3.0 however, was a major, backwards incompatible release, which was published in 2008. It is the successor of the 2.x series and will replace it completely in the near future.

Also, since the 2.x series, the Python project created a transparent and community backed development process with van Rossum as its leader.

5.3.2 *Community*

The Python project has a large user and developer community. Many are working on the CPython interpreter and the standard library, however there are also a lot of other current Python related projects, such as advanced libraries.

The Python community is very eager to attend and offer opportunities to meet in person. Therefore, one can frequently attend conferences and workshops worldwide [Pyt#1]. The first conference was *PyCon* in North America in 2003. However there are now many other conferences, such as *PyCon DE*, *EuroPython* or *SciPy*.

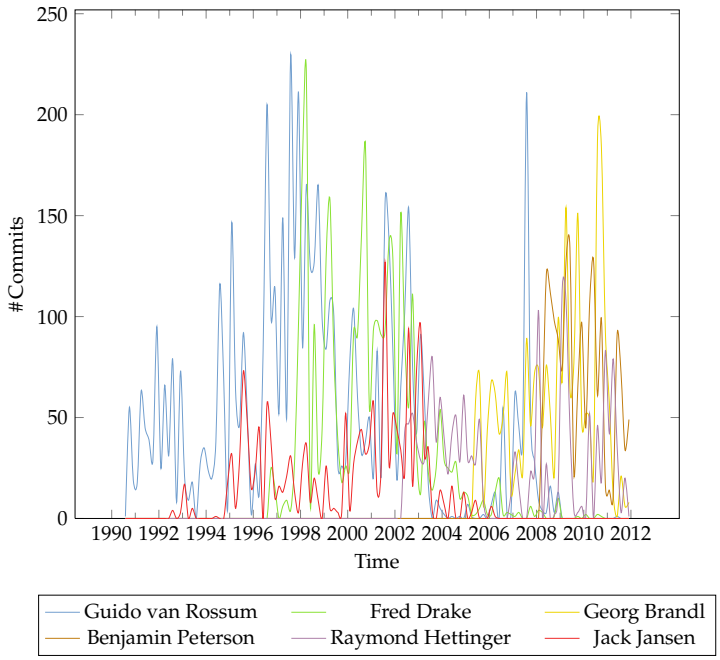


Figure 5.15: Monthly activity of the most active CPython developers. The project leader Guido van Rossum appears to be busy with his position as other developers such as Georg Brandl are much more active.

The communication inside the Python project mostly takes place through several mailing lists [Pyt#3]. Each mailing list stands for a certain topic, for example the *python-dev* mailing list which is the main communication channel for Python development. The *python-ideas* mailing list on the other hand is for future ideas and goals of the Python project. There are of course other communication methods available, such as several IRC channels and developer's blogs.

Next to the already mentioned users and developers of Python people will be classified into several categories depending on how strongly they are involved in the development of Python [Pyt#5].

DEVELOPER One is able to gain the developer role when they have consistently shown contributions to Python. There is however no set rule of how many patches or resolved issues are needed. It is possible to request the developer role by asking any other person, who already has the developer role. They then will decide if the requester is ready to gain the additional privileges. The enquired person will sequently act as a mentor for the requester.

CORE DEVELOPER The way to become a core developer is quite similar to the developer role above. One must have consistently contributed patches, which meet a certain quality standard. Then normally a developer will be offered the chance to become a core de-

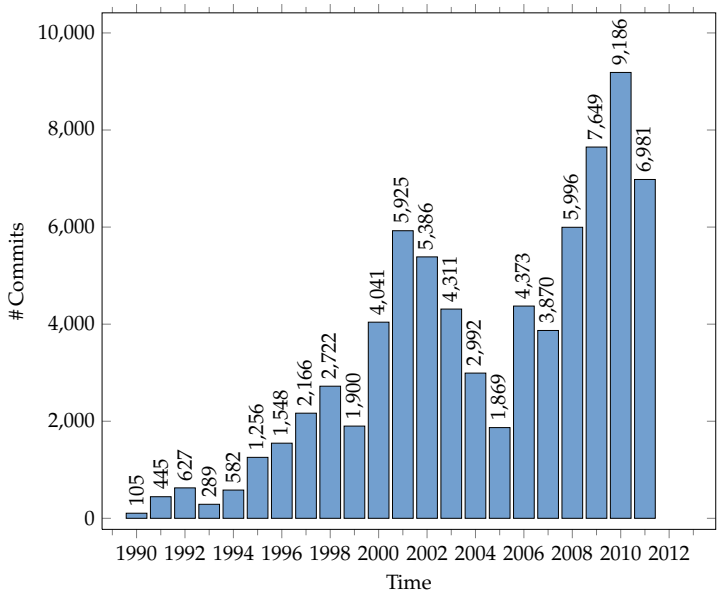


Figure 5.16: Yearly overview of commits to CPython. The two peaks appear to match the releases of the major releases Python 2 and Python 3.

veloper. The reference person will also act as a mentor for the new core developer. Also, the core developer group will watch if the quality of the contributed patches was met and the development process understood.

EXPERT Each core developer has the possibility to become an expert in a certain area of the Python project. An expert is the maintainer of his field of interest and therefore responsible for changes and new features to those areas. All issues, help requests and decisions will be forwarded to the expert. Experts may also be asked to decide on features or bugs.

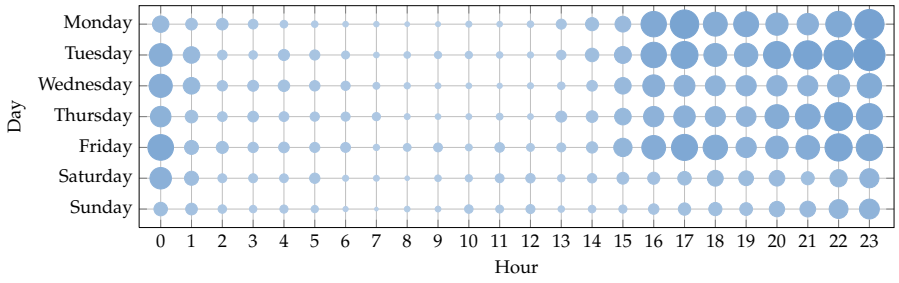


Figure 5.17: Time based view on commits of core contributors. It is interesting to note that almost all commits happen during the week, however later than usual office hours.

BENEVOLENT DICTATOR FOR LIFE (BDFL) Guido van Rossum is the only person who holds this title. As the leader of the Python

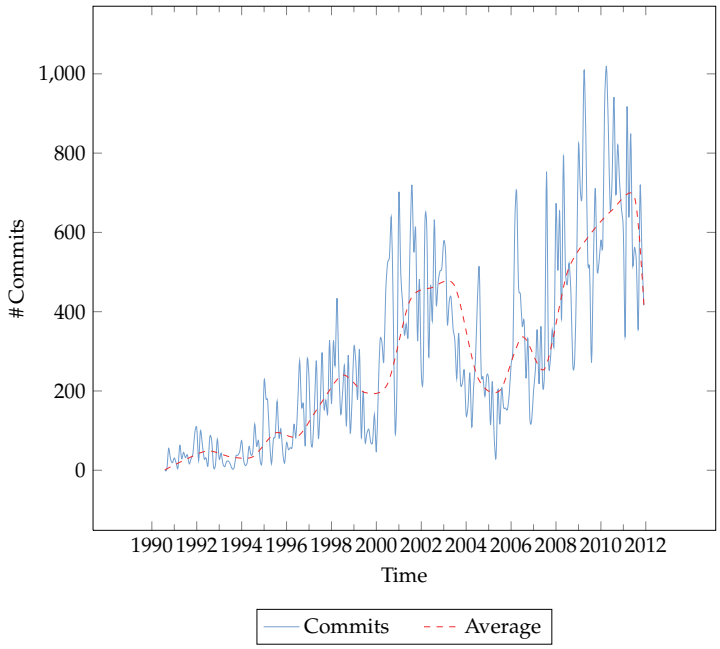


Figure 5.18: Amount of commits per month of core contributors. Again this matches with the development phases of Python 2 and Python 3.

project, he has the privilege to outvote and overrule any decision made by an expert or core developer. However his decisions should always be in favour of the project, as the word *benevolent* states.

5.3.3 *Release Process*

For releases, Python uses a version naming with three numbers. The numbers stand for major, minor and micro versions [Pyt#2; WR01]. New major releases are extremely scarce. They are planned for a long time in advance, often with incompatible changes. An example for such a major release is Python 3.0. Minor releases are feature releases with no incompatibilities between its predecessors. Roughly, they get published every 18 months. Micro releases are bug fix releases and get promulgated every six months, although they can be released in shorter time periods too. Besides, each release is proceeded by several alpha, beta and release candidate releases which are testing releases.

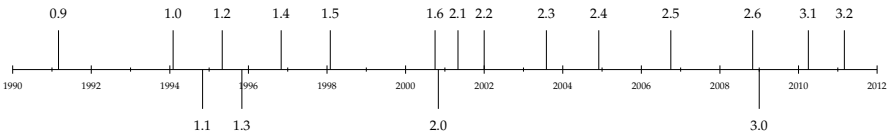


Figure 5.19: Major releases of Python.

PEP: 8
Title: Style Guide for Python Code
Version: 00f8e3bb1197
Last-Modified: 2011-06-13 12:48:33 -0400 (Mon, 13 Jun 2011)
Author: Guido van Rossum <guido at python.org>, Barry Warsaw <barry at python.org>
Status: Active
Type: Process
Created: 05-Jul-2001
Post-History: 05-Jul-2001

Introduction

```
This document gives coding conventions for the Python code comprising the
standard library in the main Python distribution.  Please see the
companion informational PEP describing style guidelines for the C code in
the C implementation of Python[1].

This document was adapted from Guido's original Python Style Guide
essay[2], with some additions from Barry's style guide[5].  Where there's
conflict, Guido's style rules for the purposes of this PEP.  This PEP may
still be incomplete (in fact, it may never be finished <wink>).
```

A Foolish Consistency is the Hobgoblin of Little Minds

```
One of Guido's key insights is that code is read much more often than it
is written.  The guidelines provided here are intended to improve the
readability of code and make it consistent across the wide spectrum of
Python code.  As PEP 20 [6] says, "Readability counts".

A style guide is about consistency.  Consistency with this style guide is
important.  Consistency within a project is more important. Consistency
within one module or function is most important.
```

Figure 5.20: Excerpt from PEP 8: Style Guide for Python Code.

5.3.4 *Development*

Since Python 2.0, the Python community changed their development process towards a more open and transparent approach. For each patch, the Python community can vote for or against it. They can vote +1, +0, -0, -1, whereas +1 and -1 mean acceptance or rejection and +0 and -0 mean an indifferent decision with a slight positive or negative slant [Waro2]. The voting itself takes place on either the *python-dev* mailing list or will get announced on the *python-announce* mailing list. This voting however is completely deliberative and Guido van Rossum can still approve or reject a patch even if the voting disagrees with his decision.

The development process is highly dependent on the so called Python Enhancement Proposal (PEP) [WHGoo]. Loosely modelled on the Internet Request for Comments process, they are design documents which describe either a proposed feature, a process or just provide information. They are continually discussed in the community and revised until the community reaches a consensus. PEPs can then either be approved or rejected. They are the primary way to propose new features and on approval are finalized with the commit of a patch.

The Python project defines three types of PEPs.

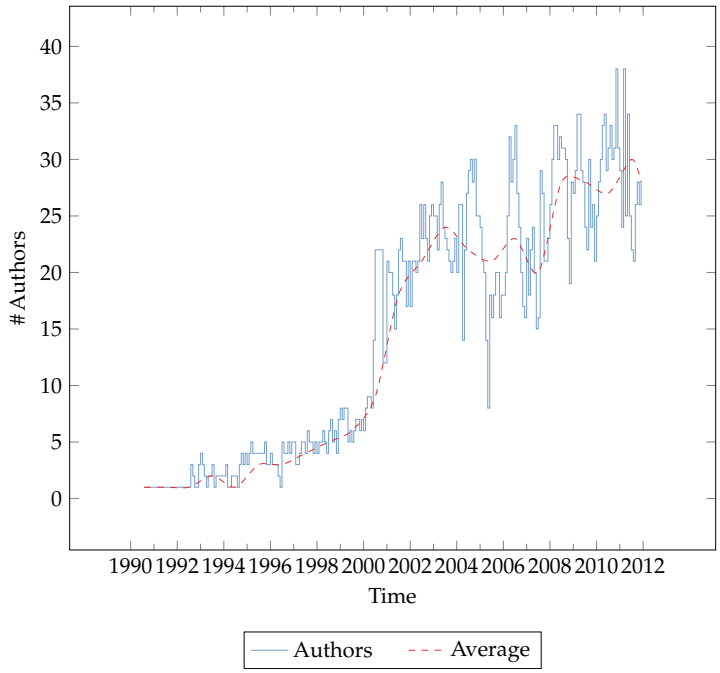


Figure 5.21: Amount of distinct authors of CPython over time. Python 2 seemed to provide a major reason for developers to join.

1. STANDARDS TRACK PEP The most common form of PEPs describes a feature proposal for the Python language. They always consist of a design document and a reference implementation.

2. INFORMATIONAL PEP Unlike the Standards Track PEP, the Informational PEP does not propose a new feature. It provides general guidelines or information about a certain issue of Python. However it does not necessarily have to reach consensus inside the Python community and so one is free to ignore Informational PEPs.

3. PROCESS PEP This kind of PEP is quite like the Standards Track PEP, except that it applies to other areas than the Python language and describes processes around Python. For example any guidelines, decision making processes, development workflows and other are mostly Process PEPs. Additionally any PEPs which describe other PEPs, for example how a PEP should look like, will be defined as Process PEPs.

The PEP process begins with a feature proposal for Python. While all bigger proposals require a PEP, the Python project suggests to submit small changes directly as a patch submission. A PEP always has one or several authors, who have the responsibility for it. They will begin by submitting the PEP to the Python project. More precisely it should be presented to the *python-ideas* mailing list and subsequently sent to the PEP editors. The PEP editors will assign a number and a cat-

egory as mentioned above. Additionally, new PEPs always start with the *Draft* status. From there, the PEP workflow begins.

DRAFT This is the default state of any new PEP on submission, as described above.

DEFERRED A PEP Editor can always, instead of assigning the Draft status, defer a PEP. Reasons for a deferral could be duplication, being poorly written, lack of motivation by the author or not being in line with the Python philosophy.

ACCEPTED A PEP will be able to get this status assigned, if all PEP criteria are matched. This includes a precise formulation of the PEP, not breaking backwards compatibility and finally be accepted by the Benevolent Dictator For Life (BDFL). However the criteria are not fully binding and a PEP just has to be accepted by the BDFL.

REJECTED At this stage a PEP can still be rejected. Mostly this status gets assigned, if it turns out that the original idea did not fit the Python project. The reject stage is just an option to drop a PEP, after it was accepted or not deferred.

WITHDRAWN A PEP can also always be withdrawn by the author. This could help if an author does not want to work on a PEP anymore, for example because there is a better solution available.

FINAL Once a PEP has been accepted and has a reference implementation available, it can get the status Final by the BDFL.

REPLACED Mostly for Informational PEPs, subsequent versions can replace a PEP. In this case the original PEP would be marked as Replaced and superseded by the new PEP.

ACTIVE Informational and Process PEPs can also be marked as Active, if they will never be completed. For example PEP 1 [WHGoo], which describes the process of PEPs has its status set to Active, as it is continually improved and adapted to new workflows.

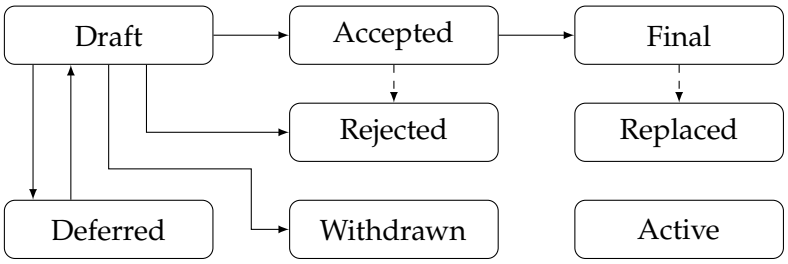


Figure 5.22: Possible paths of the status of Python Enhancement Proposals.

5.4 PHP PROJECT ANALYSIS

PHP is a server side, interpreted programming language designed for web applications and development [PHP#9]. As such, it requires a web server with a connected PHP installation. However it can also be used with a command line interface. It is widely available for all major platforms and is licensed under the PHP License [PHP#4]. It is worth noticing, that while it is a Free Software license, it is incompatible to the widely used GNU GPL. PHP comes with an extensive standard library and supports many programming paradigms, such as object oriented, imperative or procedural programming styles. The abbreviation PHP originally stood for Personal Home Page [PHP#2]. With PHP version 3, the project changed its name to PHP Hypertext Processor. PHP is a very popular programming language for web development and featuring a large number of websites [W3T#2; PHP07].

5.4.1 *History*

In 1994, Rasmus Lerdorf created the first version of PHP for his own website, naming the program Personal Home Page Tools [PHP#2]. He published the bundle one year later, after rewriting the original version. That also included the first name change to Forms Interpreter (FI) and then to Personal Home Page Construction Kit the same year. At that time, PHP could be considered as an advanced programming interface.

However, it received another makeover in 1996, combining the two previous names to PHP/FI. This time, PHP truly began to evolve to a full featured programming language. It included several modules for database or browser interaction. This version was also known as PHP/FI 2.0. Following a popularity boom in 1997 and 1998 it reached its limitations due to the design of the project and by being almost solely developed by Lerdorf.

In 1997 Andi Gutmans and Zeev Suraski rewrote the PHP interpreter and approached Lerdorf discussing the problems PHP had. Together, they rewrote and redesigned the language and also changed the name to PHP Hypertext Processor. Version 3.0 of PHP was then released in 1998 and replaced PHP/FI 2.0 completely. One of the most important features of the new approach was to provide an interface for modules, which can be used directly from the programming language.

The design of the language was changed further in 1998 improving performance and the modularity of the codebase. The new engine was called Zend Engine and provided the core of PHP 4.0, which was released in 2000.

Four years later, PHP 5 was released featuring the successor Zend Engine 2.0. This release was incompatible to the previous 4.0 version

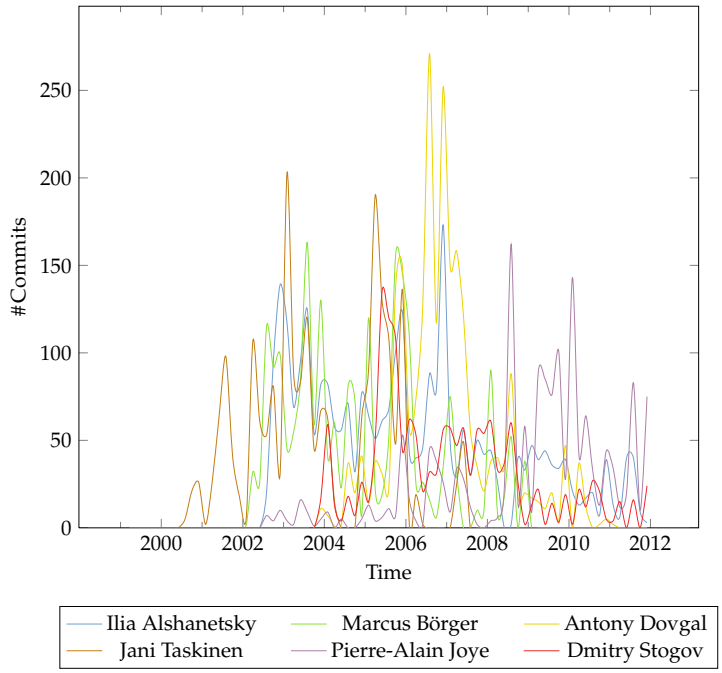


Figure 5.23: Monthly activity of the most active PHP developers. The climax of development in 2007 stalled and the most active developers appear to lose interest since then.

and a large initiative was planned and executed to promote the transition from PHP 4 to PHP 5.

5.4.2 *Community*

The PHP project consists of a large user and developer community [Mag10]. Additionally to the PHP interpreter, there are many people working on PHP extensions and components, which get distributed alongside the standard interpreter.

The PHP community provides a large amount of conferences and workshops to meet in person and work together on PHP. While the *International PHP Conference* was the first of its kind in 2001 [PHP#3], there are many opportunities to meet other PHP contributors. A lot of local PHP User Groups exist and provide weekly or monthly meetings and meet-ups such as workshops.

Most of the communication inside the PHP project takes place through mailing lists [Mag10]. There exist mailing lists for almost all aspects of the project, however the most important is the *internals* mailing list. Most of the development process, future ideas and new contributions are handled through that list. Of course, several IRC channels and blogs of PHP developers are also used.

Officially, there is no categorizing of developers and everyone is treated equally. However the project uses a concept named *karma*, which means that one increases his karma by contributing to the pro-

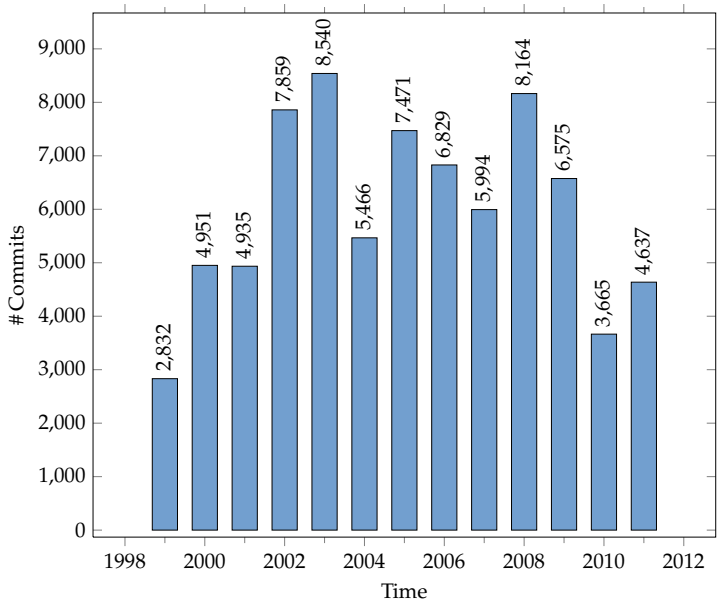


Figure 5.24: Yearly overview of commits to PHP. The peaks could match with the development and releases of PHP 4.3 and PHP 5.

ject with code, discussions and new ideas [Mag10]. Furthermore, several people are responsible for different parts of the code or modules and can be classified using those criteria [PHP#1].

DEVELOPER One is called a PHP developer, once they get a PHP Subversion account. According to the PHP community, an account needs to be earned, which means that one has to provide several patches and contributions to the PHP project first. Once one has shown enough commitment, they can apply for an account. The account however will only be granted if a reference person approves.

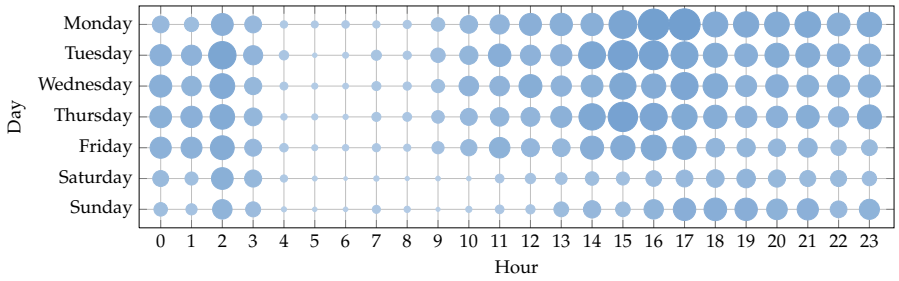


Figure 5.25: Time based view on commits of contributors. A lot of the commits appear to be happened during work days and times with some volunteer commits in the evenings and Sundays.

MODULE AUTHOR Each developer has the possibility to become a module author. This role has the responsibility for a certain PHP module, which gets distributed with the PHP release. Examples for such

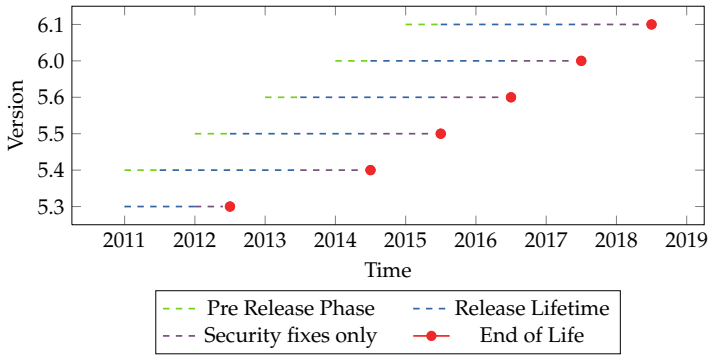


Figure 5.26: Preliminary PHP release cycle.

modules are database or imaging modules. For each module, there can be more than one module authors, who share the responsibility.

PHP AUTHOR This role defines a certain section of the PHP interpreter and its responsibility for it. It is quite similar to the module author role, except that it covers only PHP itself.

LANGUAGE DESIGNER Currently only Rasmus Lerdorf, Andi Gutmans and Zeev Suraski hold this role. They are the only who can actively change the programming language syntax and concepts.

5.4.3 *Release Process*

In 2010, the PHP project set up a new release plan with detailed information about how the release process should work [PHP#6]. Before, individuals decided when a release happened and which features it included. The new release cycle features two release managers who are voted by PHP developers. The PHP project uses a versioning scheme with three numbers for releases: major, minor and micro. Major releases get published quite seldom, can break backwards compatibility and are planned a long time in advance. Minor releases can have new features and bug fixes, however backwards compatibility must be kept. Micro release are bug fix only releases where backwards and API compatibility must be kept. Each major release is followed by a number of minor releases, which themselves are followed by a number of micro releases.

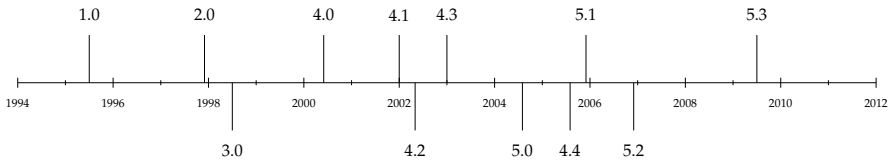


Figure 5.27: Major releases of PHP.

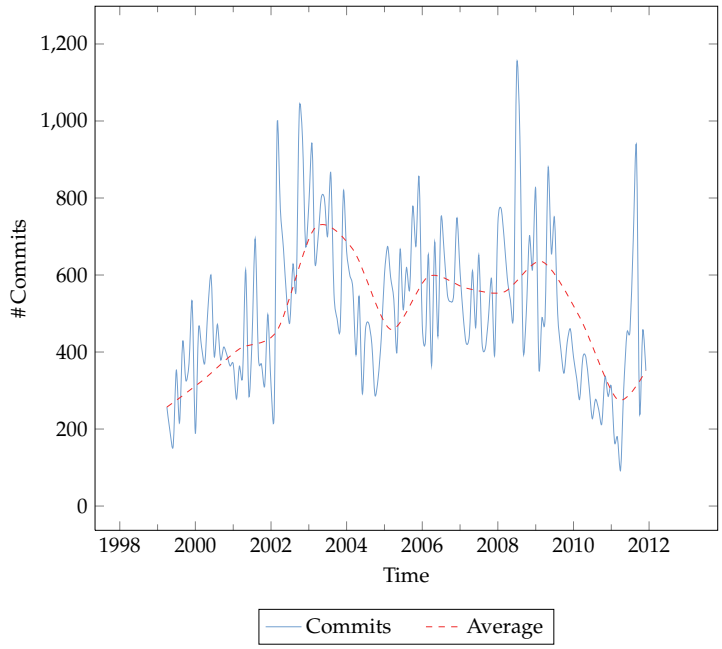


Figure 5.28: Amount of commits per month of core contributors. Even if the previous graphs showed a stalling development, the recent happenings in 2012 appear to show an again growing project.

Starting with PHP 5.4 a major or minor release will get published each year. Each yearly release is then followed by a number of micro releases for two years, which only contains bug fixes. After that period, only micro releases with security relevant fixes will get published.

5.4.4 *Development*

Most of the development process is handled through the PHP mailing lists [PHP#6; Mag10; PHP#7]. Not every patch however needs to be discussed and approved first, minor features or changes often get directly committed to the repository. Every change needs to pass a peer review process, where other developers review the made changes. As well every change gets reviewed by other developers, who in case the need arises, discuss the change on the developers mailing list. Generally the result is often a more comprehensive change or new features.

To help the development and decision process, the PHP project keeps track of new ideas, features and proposals through so called Request for Comments (RFC) [PHP#5]. Those are documents which describe the new feature and its rationale. A RFC document is always owned by at least one person, who is responsible for it. The RFC process begins with the author's submission of the RFC to the PHP wiki and an announcement to the *internals* mailing list. Depending on the importance and affected sections of the PHP project, the following

discussion period is set to a minimum of one to two weeks. It can be longer however, but not shorter [PHP#7].

After the discussion period has passed, the author can either call for a vote or extend the discussion period as needed [PHP#7]. The vote is announced on the mailing list and followed by a voting period which should be at least one week, but can be extended if required.

Depending on the importance of a RFC, it takes either 2/3 of all votes or 50 % plus one to get accepted. The importance is defined by whether it actually changes the syntax or behaviour of the PHP language and is therefore irreversible or not. A failed proposal can be resurrected at earliest six months after the last vote or if the author makes considerable changes to the RFC. There is no definition by the PHP project, what considerable changes are, however the opinion is that it should be changed in a way, that it significantly influences a vote's outcome.

All PHP developers are allowed to vote and, if needed, representatives from the PHP community, such as participants of PHP related discussions or developers of PHP appendant projects can participate in the voting process. Those however have to be chosen by PHP developers [PHP#8].

During the whole development process, a RFC has an assignment status, which can be one of the following [PHP#5].

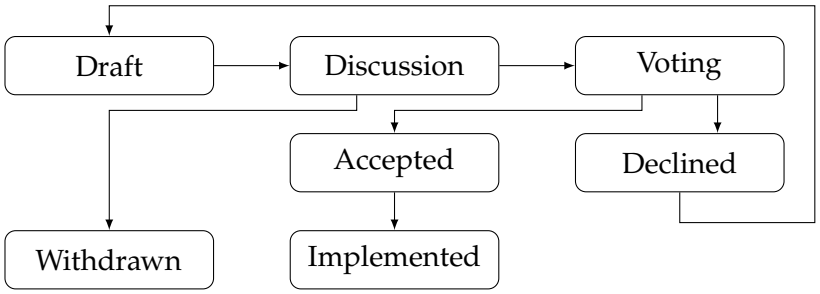


Figure 5.29: Possible paths of the status of Request for Comments.

DRAFT Once a RFC gets submitted to the PHP wiki, it gets this status assigned. The Draft status means, that the RFC is not ready yet for a discussion, however it can be improved also by other people and a preliminary discussion can be started.

DISCUSSION Once the author of a RFC announces it for discussion, it gets assigned to this status. As previously described, the discussion phase starts with this status.

VOTING After the discussion phase the voting begins. While developers can vote for or against the RFC, the voting status gets assigned including a link to the voting site.

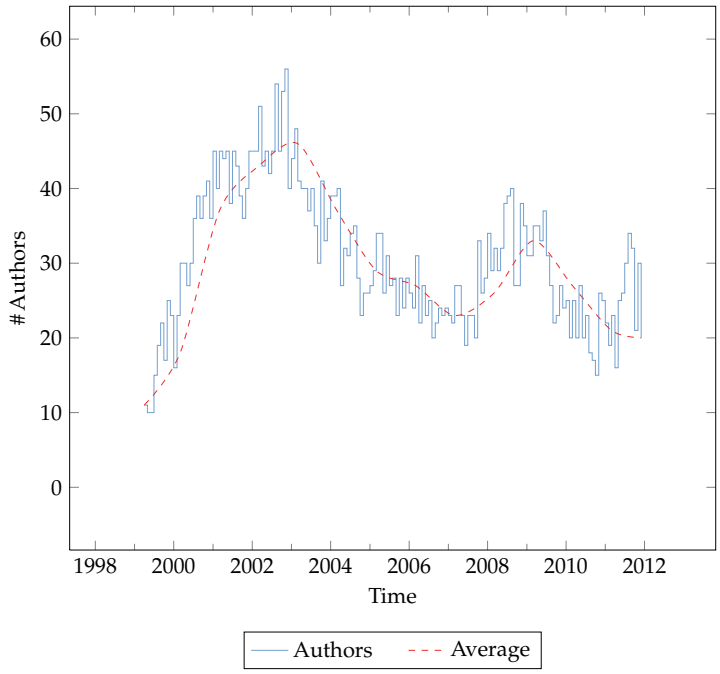


Figure 5.30: Amount of distinct authors of PHP over time. The number seems to be stable between 20 and 30 authors.

ACCEPTED Once a voting was successful, the RFC is accepted and can be implemented.

DECLINED If the majority voted against the RFC, it gets declined. After six months and several changes to it, the author can resubmit it.

IMPLEMENTED If the voting was successful, the RFC will be implemented or taken into action if it describes a community process.

WITHDRAWN The author can withdraw the RFC without calling for a vote if they think the RFC is irrelevant or won't pass the voting.

5.5 GNOME PROJECT ANALYSIS

GNOME is a desktop environment for UNIX based systems. It is composed by a collection of tools and programs including a desktop shell in order to provide all the essential utilities a user might need when working with a computer. Officially, GNOME is part of the GNU's Not Unix (GNU) project and licensed under the GNU GPL and the GNU Lesser General Public License (GNU LGPL). The name GNOME was initially an acronym for *GNU Network Object Model Environment*, however that acronym was dropped. The GNOME project targets ease of use and user friendliness and therefore aims for coherent and good user interfaces [GNO#5], accessibility, internationalization, regular re-

leases and good support for users and developers. GNOME is a modular project, meaning that it consists of several so called modules, which can be either applications, libraries or utilities. Since the release of GNOME 3, the modules were reorganized into a GNOME Core suite and a GNOME Apps suite. GNOME Core provides everything to run a basic desktop system and will therefore be analyzed in this context.

5.5.1 *History*

GNOME was first announced and started in 1997 by Miguel de Icaza and Federico Mena Quintero as a counterpart to KDE [Ger03; GNO#2; GNO97]. Both were university students at the time when they set their aim to produce a desktop environment using only FOSS technologies. KDE relied on the Qt widget toolkit, which at the time was licensed under a proprietary software license. Instead of using Qt, they used the GIMP Tool Kit (GTK) originally developed for the GIMP graphics editor. The GNOME project quickly grew into a large project which nowadays is the most popular desktop environment for UNIX type operating systems. The desktop as well as the developer technologies can be found on workstations and large enterprises but also on mobile devices. With the recent GNOME 3 release a major overhaul with a significant redesign of the desktop environment and an entirely new user interface took place [GNO11].

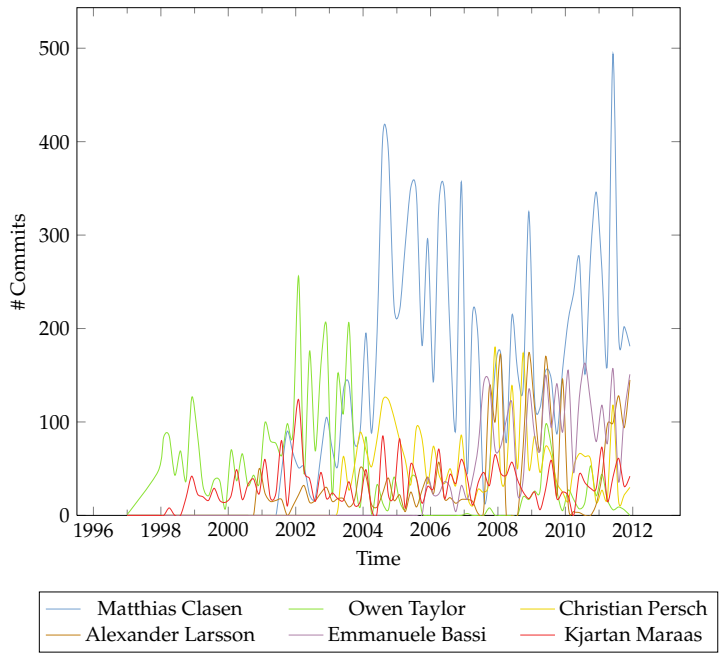


Figure 5.31: Monthly activity of the most active GNOME Core developers. The disproportional amount of commits by Matthias Clasen can be interpreted with a high skill set and lots of effort he puts in the project.

EVENT	VENUE	DATE
GUADEC I	Paris, France	2000
GUADEC II	Copenhagen, Denmark	2001
GUADEC III	Seville, Spain	2002
GUADEC IV	Dublin, Ireland	2003
GUADEC V	Kristiansand, Norway	2004
GUADEC VI	Stuttgart, Germany	2005
GUADEC VII	Vilanova i la Geltrú, Spain	2006
GUADEC VIII	Birmingham, England	2007
GUADEC IX	Istanbul, Turkey	2008
Desktop Summit	Gran Canaria, Spain	2009
GUADEC X	The Hague, Netherlands	2010
Desktop Summit	Berlin, Germany	2011
GUADEC XI	La Coruña, Spain	2012

Table 5.2: Previous and planned GNOME conferences.

5.5.2 *Community*

The GNOME project consists of a large user and developer community. While there were about 3500 people contributing to GNOME and its applications, there is also a big community around the project, which for example uses GNOME technologies such as GStreamer or GTK [GNO#2; GNO#10].

Most of the communication inside the project is handled through mailing lists and IRC channels. Almost every GNOME module has a dedicated mailing list or IRC channel. Global decisions are mostly handled through the *desktop-devel* mailing list. Additionally blogs are widely spread in the GNOME community and contributors often write blog posts about achievements, wishes or start discussions.

Next to several hackfests each year, the community holds a yearly conference under the name GUADEC. It stands for GNOME User and Developer European Conference, and while the conference only took place in Europe so far it is nevertheless considered worldwide by the community [GNO#1].

Due to the large and modular composition of the GNOME project, there are similar roles for each module. Only very few teams and roles stand above all modules. In this respect the developer community can be very much be seen as a flat structure [GNO#10; Ger03; GNO#3; GNO#7].

COMMITTER Any person who contributed a reasonable amount of improvements to the GNOME project or to a single module can be-

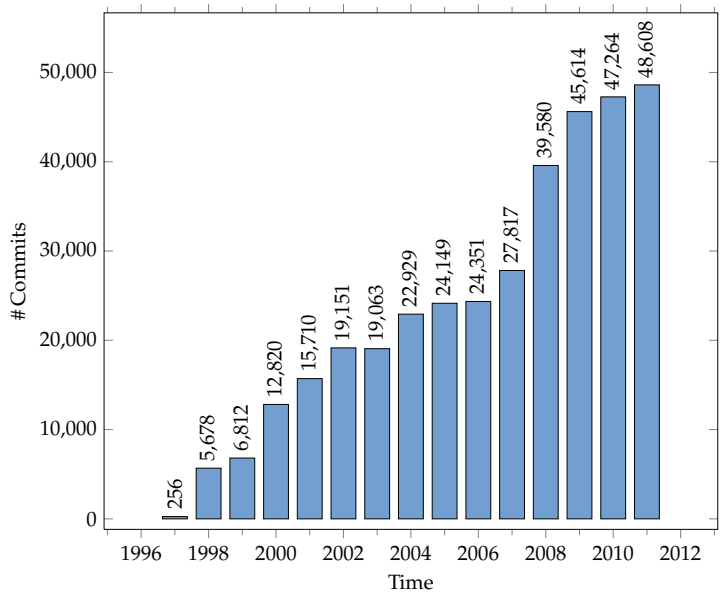


Figure 5.32: Yearly overview of commits to GNOME Core. The leap in 2008 appears to be related to GNOME 2.22 or GNOME 2.24.

come a committer. A committer has full read and write access to all GNOME repositories. Each commit will however be reviewed and approved by the maintainer of the module. To get such an account, one has to make a formal request to the accounts team along with one or several vouchers who can confirm ones contributions to the GNOME project. Every module maintainer or translation team leader can act as a voucher. Furthermore the voucher will be responsible for the actions of the requesting person on the repositories.

MAINTAINER Every module has one or more maintainers who will be responsible for releases, reviewing patches and the direction of a module. They are the main contact for the community and therefore act as the leader of a certain module. A single module can be administered by multiple maintainers and one maintainer can administer several modules. To become a maintainer one has either to create a new module which gets incorporated into the GNOME project or be asked by another maintainer.

RELEASE TEAM This team is responsible for a wide range of tasks concerning the development process of the GNOME project. Their tasks include for example creating a development schedule, making and publishing releases, approving or rejecting freeze break requests and defining the module list for GNOME releases. The team size is not fixed and can vary over time. Membership is only by invitation and often only when one person leaves the team to free up one space. The leaving person may recommend a new team member which the team will decide upon.

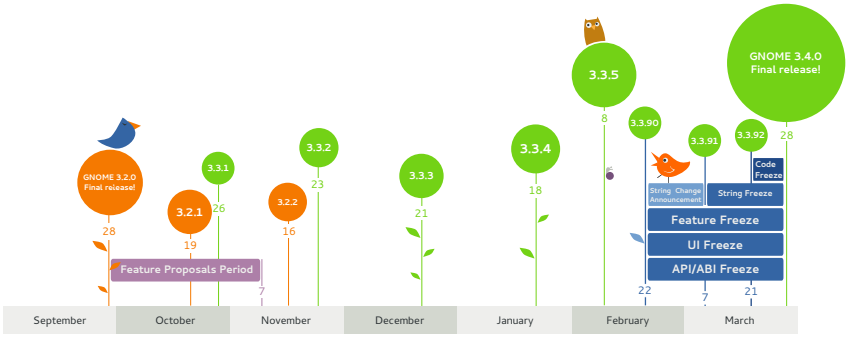


Figure 5.33: Release schedule which will lead to GNOME 3.4.

DESIGN TEAM Since GNOME 3 the design team holds a much more important role in the GNOME project as they primarily define the design of the GNOME user experience. This is done by designing the user interface and workflow of GNOME modules. Additionally they play an important part in the feature based development process.

GNOME FOUNDERS Miguel de Icaza and Federico Mena Quintero, the original GNOME founders, played an important role in the first years of the GNOME project. Nowadays however they are more active in projects around GNOME technologies and often act as visionaries.

5.5.3 *Release Process*

The GNOME project uses a versioning scheme with three numbers. However, it only distinguishes between major and minor releases [GNO#4; GNO#6]. The numbers are however nevertheless called major, minor and micro numbers. An incrementation of the first number only occurs when the project does a ground-breaking change, such as using GTK+2 for GNOME 2.x or the new user experience with the GNOME shell and GTK+3 for GNOME 3.x. For the minor version number the GNOME project uses odd numbers indicating an unstable series and even numbers for stable releases. For example the unstable 3.1.x series will become the stable 3.2.x release series.

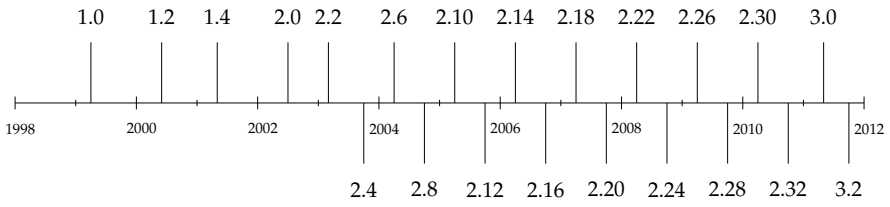


Figure 5.34: Major releases of GNOME.

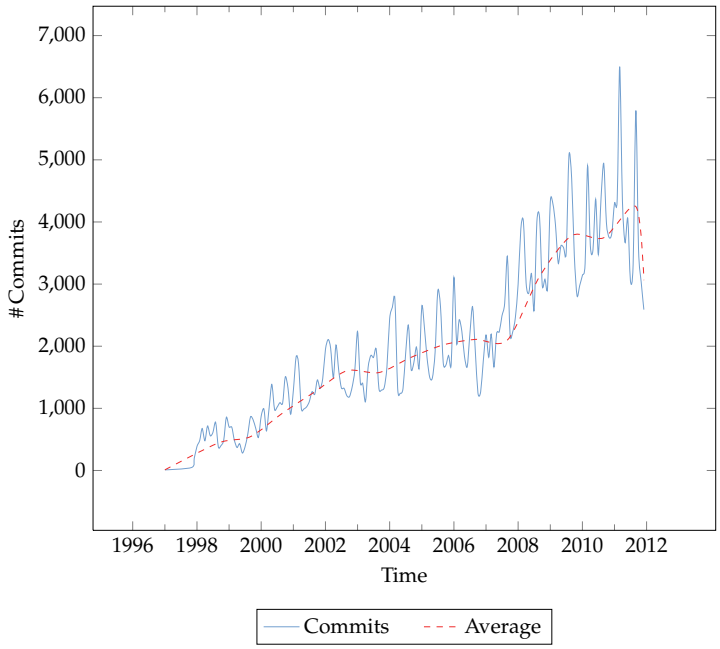


Figure 5.35: Amount of commits per month of core contributors. The project quite shows a linear growth with peaks every six months.

The release schedule is fixed with a new major release appearing every six months. To achieve this, the release team publishes a release schedule in the same frequency [GNO#4]. Over the years it stayed mostly the same, while few minor changes may occur to comprise conferences or holidays. The release schedule includes future stable minor releases, and so the current and future schedule overlap.

Each stable release series consists of at least three releases which are named with x.y.0 to x.y.2. Of course, a module maintainer can decide to provide more stable releases [GNO#7]. The unstable series consists of eight releases which are distributed over six months. The schedule furthermore includes proposal and freeze periods which will be described in the following [GNO#4; GNO#6].

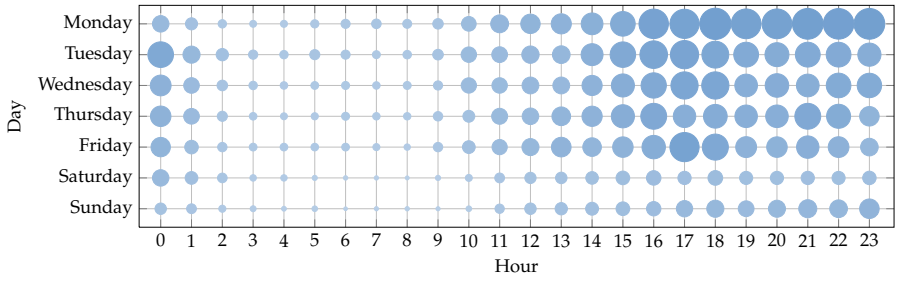


Figure 5.36: Time based view on commits of core contributors. It seems that most developers are employed. The high number on Monday can be explained with the release preparation which are always done on Monday evening.

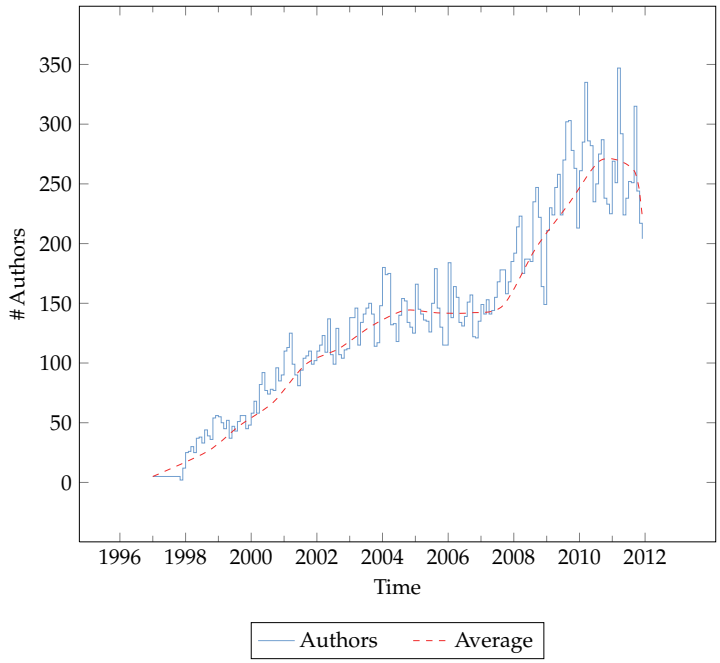


Figure 5.37: Amount of distinct authors of GNOME Core over time. Especially the development of GNOME 3 which started in 2010 appears to have attracted additional developers.

FEATURE PROPOSAL PERIOD After a major release gets published the feature proposal period starts. GNOME developers can propose features for the next major release and discuss them with the community. Approximately a month later, when the first unstable release gets published, the proposal period ends. Around two weeks later the release team meets and decides about proposed features with the community input given up to this point.

THE FREEZE With the first beta released the first freeze takes action. No user interface changes, new features or developer API changes are allowed without approval from the release team. This excludes of course bug fixes. Additionally new translatable strings must be announced to the translation and documentation team.

STRING FREEZE With the second beta release no string changes are allowed anymore without confirmation of both the release and translation team.

HARD CODE FREEZE With the release of the release candidate no source code changes are allowed without approval of the release team. Documentation and translation however can continue. After publishing the next major GNOME release the Hard Code Freeze ends but all other freezes remain in action for the stable series.

5.5.4 *Development*

Before the release of GNOME 3, the release team together with the community decide about the inclusion of new modules. The development of each module was planned and executed by the maintainers. Since GNOME 3, the development workflow changed into a more design driven development approach. This means that all new features concerning user interfaces or applications have to go through the design team [GNO#3]. This leads to a consistent process with a uniform user interface. However the design team has no decision making authority and a maintainer can ignore the design team's proposal.

Next to the design driven approach the module inclusion changed to a feature inclusion approach [GNO#8; GNO#9]. Every GNOME contributor can create a feature proposal in which one describes a feature they would like to see in the next stable GNOME release. A feature proposal is composed by a description of the problem to be solved, one or more authors, a list of involved parties and the current state of the proposal. All of these fields are required and a feature will not be accepted until all are completed. After each major release and until about a month later, one can propose a feature for inclusion. The community has time to discuss the features and improve them for roughly one and a half months. After that time the release team will meet and depending on the feedback of the community decide about the inclusion of each feature. If a feature gets accepted the author of the feature will be its leader and responsible for the completion. With the release of the first beta release and the establishment of the Feature, User Interface (UI) and API/ABI Freeze the feature has to be finished and working. If not, it may be postponed to the next major release.

5.6 KDE PROJECT ANALYSIS

KDE is a desktop environment designed to run on UNIX based systems as well as Microsoft Windows and Mac OS X systems [KDE#9; KDE#1]. Since 2010 and the release of KDE 4.4 the project is known as KDE Software Compilation (KDE SC). It is composed by the desktop environment named Plasma Desktop and several core applications for the daily needs. KDE SC is licensed under the GNU GPL and GNU LGPL. The name was originally an acronym for Kool Desktop Environment and later for K Desktop Environment, however it is no longer in use. KDE is a modular project consisting of several libraries which allow developers to build their programs around the platform. As the KDE project and the KDE SC stand for a wide range of technologies and applications, this analysis will be limited to the core of the KDE SC, also known as KDE Base.

5.6.1 *History*

KDE was first announced by Matthias Ettrich in 1996 as a desktop environment for end users with the same look and feel for all applications [KDE96]. The name KDE was a word play with the existing Common Desktop Environment (CDE), which was highly popular at that time. Matthias Ettrich chose the Qt framework as the graphical library, which was and is developed by the company Trolltech. KDE 1.0 was then released in 1998 [KDE#7]. At the same time, Trolltech dual licensed the framework under the Q Public License (QPL) and a proprietary software license. However, it was debated if that license was compatible with the GNU GPL and in 2000 the framework was licensed under the GNU GPL which ceased the criticism.

In 2009 the KDE project renamed the project to KDE SC and redefined the project as a community which delivers FOSS for user interfaces by emphasizing the KDE technologies [KDE#8]. All software created with KDE technologies are KDE projects. However KDE SC only contains projects which derive from the project itself and share a common release cycle.

KDE SC 4 was a new approach to the desktop metaphor and created a new user experience for users. The centerpiece is the Plasma Workspace which exists for several devices, such as for desktop computers, netbooks, tablets and smartphones.

5.6.2 *Community*

The KDE project is one of the largest FOSS projects and according to the community the second largest after the Linux Kernel [KDE#9]. There are about 1800 active contributors to the project and its surroundings and is used by over a million people. Also, the community spans around projects based on KDE technologies.

Most of the communication in the KDE project takes place via mailing lists, most importantly *kde-devel* and *kde-core-devel* [KDE#10; KDE#2]. While the first one is mostly used for communication by application developers, the latter one is for communication on the KDE core project. Furthermore communication can happen over IRC channels, blogs or forums.

The most important KDE conference is the annual *Akademy* which is held at different locations in Europe [KDE#7]. The first KDE conferences were however named after the current release which started with *KDE One* in 1997. That is also the reason, why the conferences did not take place annually at that time. In 2004 it changed the name to Akademy. Beginning with 2009 the project incorporated the conference into the *Desktop Summit* in collaboration with the GNOME project.

EVENT	VENUE	DATE
KDE One	Arnsberg, Germany	1997
KDE Two	Erlangen, Germany	1999
KDE Three Beta	Trysil, Norway	2000
KDE Three	Nürnberg, Germany	2002
Kastle	Nové Hrady, Czech Republic	2003
aKademy	Ludwigsburg, Germany	2004
aKademy	Málaga, Spain	2005
aKademy	Dublin, Ireland	2006
aKademy	Glasgow, Scotland	2007
Akademy	Sint-Katelijne-Waver, Belgium	2008
Desktop Summit	Gran Canaria, Spain	2009
Akademy	Tampere, Finland	2010
Desktop Summit	Berlin, Germany	2011
Akademy	Tallinn, Estonia	2012

Table 5.3: Previous and planned KDE conferences.

Due to the size of the KDE project it is organized around many independent teams with a communication structure between them. There are few groups who stand above those teams and only take care about coordinating the project [KDE#3; KDE#10].

CONTRIBUTOR After having contributed to any KDE project for some time and planning to contribute in the future, a KDE contributor account can be requested [KDE#2; KDE#5]. In the request one must state why one is interested in having an account. The KDE sysadmin team will then check the application and grant an account or not.

MODULE COORDINATOR As the KDE project is highly modular, each application or library has an independent team with its own structure besides a few exceptions. A module coordinator plans and executes releases and the direction of a single module. As the teams are so diverse, there are many ways to become a module coordinator. The most obvious is of course by unweary contributions to a project.

CORE TEAM One of the most important teams is the core team as it defines the overall direction the project is heading [KDE#10]. There is no single person inside the team responsible for decision making, instead the team discusses the issue and comes up with a solution together. The primary communication method is the *kde-core-devel* mailing list which is publicly readable, however an approval is

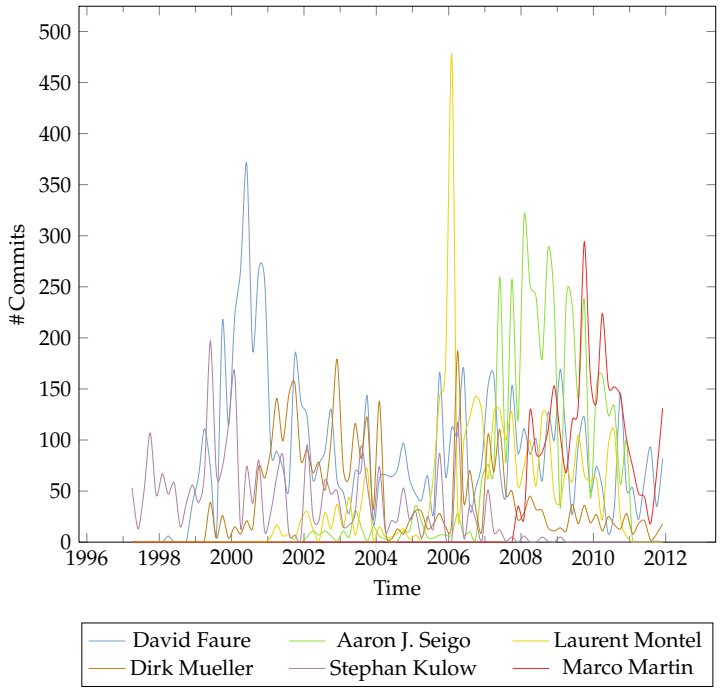


Figure 5.38: Monthly activity of the most active KDE Base developers. The most active developers almost switched completely when comparing the inception phase to nowadays.

required to join it. The KDE project does not define clear restrictions to membership and therefore a membership is granted by invite only after distinguishing or outstanding work for the KDE project.

RELEASE TEAM The actual release and the release schedules are provided and enforced by the release team [KDE#11]. It is composed by module coordinators and other release team members who actually implement the releases. The release team makes sure that all modules are following the release schedule and are in a good shape. Also they decide on code freeze breaks and take decision about future features of a release.

KDE FOUNDER Matthias Ettrich, the founder of the KDE project is no longer actively involved in the project as he works full time on the underlying Qt framework. However he has still an important voice in the community and contributes indirectly to KDE by his role in the Qt project.

5.6.3 *Release Process*

The KDE project uses a versioning scheme with three numbers and distinguishes between major and minor releases [KDE#11; KDE#6; KDE#12]. The numbers are called major, minor and micro numbers.

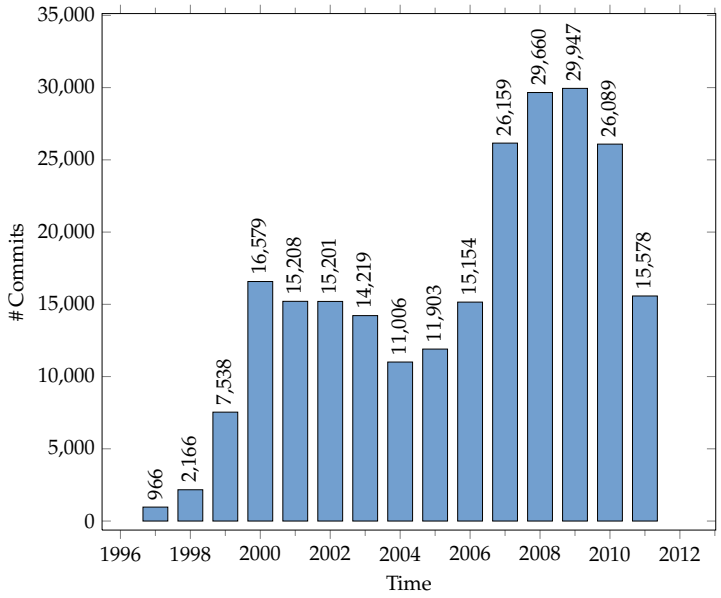


Figure 5.39: Yearly overview of commits to KDE Base. The development and release of KDE 4 and following releases in 2008 certainly attracted a lot of attention.

The major and minor numbers define major releases while the micro number defines maintenance releases. The release is a platform or standard release depending on whether the major or minor numbers change. A platform release defines a new series of releases which can break backwards compatibility to the previous platform release. They are often planned a long time ahead, usually including changes of used libraries or used library versions. A standard release however has to maintain backwards compatibility with its series. They can have new features and user interface changes, however a KDE application has to work on all releases of the same platform. Maintenance releases are not allowed to come with new features, except bug fixes or small enhancements. Beginning with KDE SC 4 the cycle changed to a major release every six months and a maintenance release roughly every month except when a major release comes out.

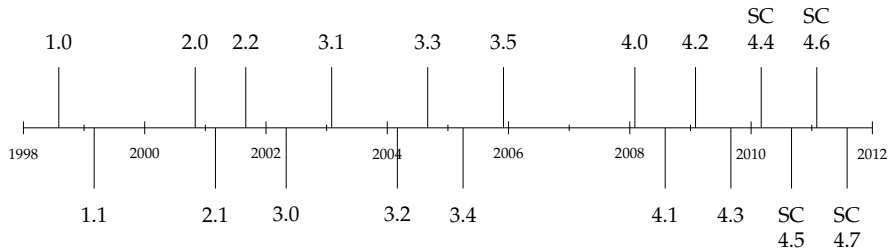


Figure 5.40: Platform and standard releases of KDE.

To ensure the release schedule, the release team publishes a new release schedule after each major release. The schedule includes all

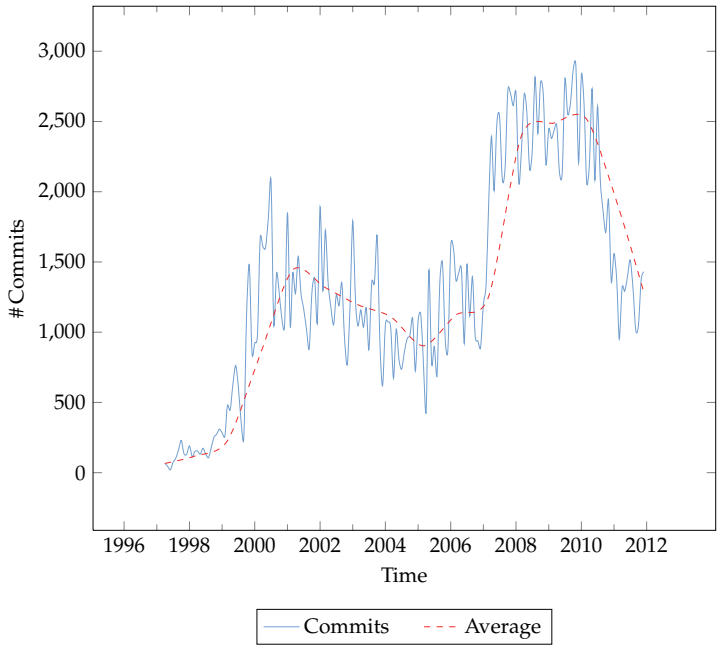


Figure 5.41: Amount of commits per month of core contributors. Again, the development phase of KDE 4 and following releases is quite visible.

deadlines and freezes which will lead to the next major release and includes future maintenance releases. This means of course that two release schedules will overlap during the month before the new major release. The following freezes and releases are provided and enforced by the release team [KDE#6].

SOFT FEATURE FREEZE Approximately three weeks before the first beta release no new features are allowed except already approved and planned features. Not finalized features have to be postponed to the next major release.

DEPENDENCY FREEZE Approximately two weeks before the first beta release no new or additional dependency versions are allowed anymore. However it is possible to request an exception by the release team.

SOFT MESSAGE, SOFT API AND HARD FEATURE FREEZE A week before the first beta release no new strings and changes to the existing messages can be made except corrections. Additionally the existing API should be almost finished and if changes are made they should be reported to the relevant teams. Lastly no new features can be added to the projects, even if they were planned for this release cycle.

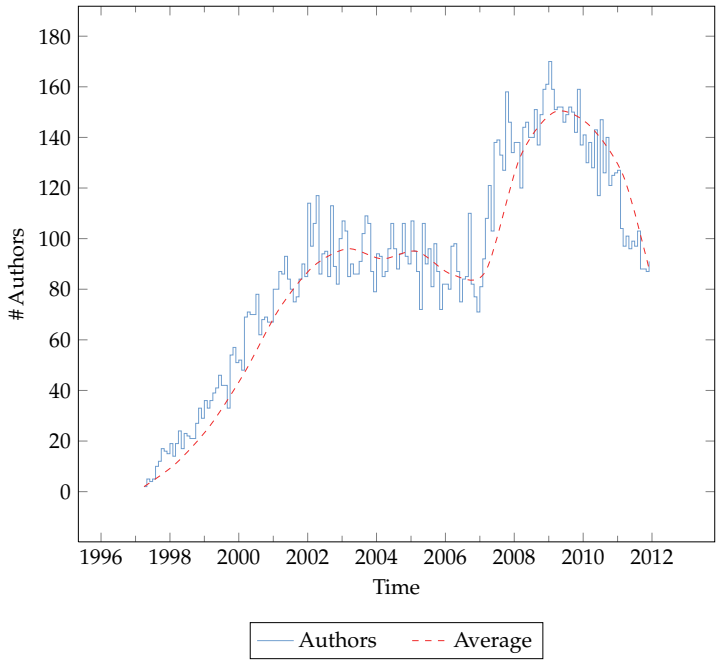


Figure 5.42: Amount of distinct authors of KDE Base over time. The project appears to grow quite linearly with a small down before the start of the KDE 4 development and the appropriate boost during and after the development.

BETA RELEASE Approximately two months before the major release the first beta release gets published. This is usually followed by a second beta release two weeks later.

HARD API, MESSAGE AND DOCUMENTATION FREEZE Five weeks before the major release the API, translatable messages and the documentation must be finished and can't be changed anymore.

RELEASE CANDIDATE Approximately a month before the final release the first release candidate will be published. Two weeks later the second release candidate follows.

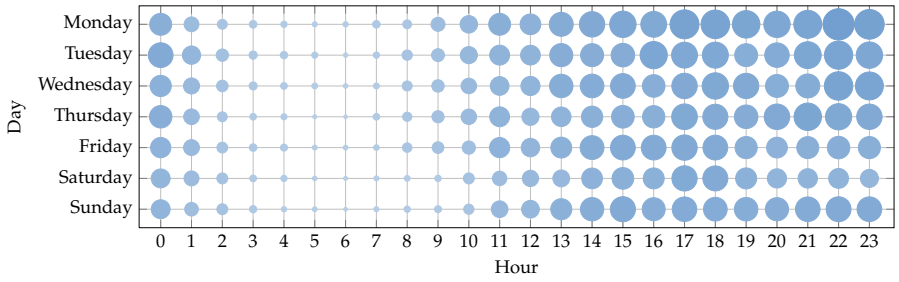


Figure 5.43: Time based view on commits of core contributors. The times are almost equally distributed probably meaning that there exist a lot of employed and volunteering developers in the community.

FINAL RELEASE Three weeks after the last release candidate the final release will then be published.

MINOR RELEASES A new minor release will be published each following month until the next major release gets published. This normally leads to four minor releases.

5.6.4 *Development*

During the development of a new major release, new project goals are set using feature proposals [KDE#3; KDE#4]. This normally happens after each major release when KDE developers have the time to propose new features for the next major release. Each feature must have an author who is willing to implement it. This often limits the number of features to already contributing KDE developers. A KDE developer is free to implement the feature by himself, however the process is often accompanied by a discussion with the specific team and several adjustments to the implementation.

During the soft feature freeze it is decided whether the feature will be available for the upcoming major release or if it will be postponed to the next major release. If the implementation already started and is almost finished, it can continue. Otherwise it will be postponed. This development process is a quite deliberate approach where people are free to implement what they see missing in the project.

5.7 PROPOSITION FOR A PROJECT ANALYSIS CATALOGUE

To characterize the development processes of FOSS projects, the following catalogue was established based on the previously gathered data and analysis. By covering all listed points a project should have been analyzed thoroughly and lead to a reasonable breakdown of all projects.

5.7.1 *Description of the Project*

A general description of the project provides information on the goals and current state of a project. They are essential to put a project under the right light and make it comparable with similar projects.

5.7.2 *Project Category*

In order to provide methods to compare projects, the project category can help to contrast projects of the same and other categories. As such it doesn't provide too much information about the project itself, but is useful for later comparisons with other projects.

5.7.3 *Scope of Analysis*

As FOSS projects often are hard to analyze due to their size and unclear definition of modules, the scope will be limited to a well known subset of the whole project. This helps to provide a thorough analysis without leaving things unattended.

5.7.4 *License*

A project cannot be counted as a FOSS project without being licensed under a FOSS license. Listing all used licenses can ensure this fact. However it is also interesting to see whether development models and communities differ if they are using a different FOSS license.

5.7.5 *History*

A project's history shaped the community and the structure of a project to the state it has today. As such the analysis of the history can bring up interesting findings.

FOUNDERS In some projects, the original founders are still present and have an influential voice, in others the original authors are no longer involved. To analyze this further, the authors have to be introduced and their role will be analyzed in the community part.

PROJECT AGE Projects always need time to evolve and find their best practise development process. As this takes time, the project age can give some information in what development process state this project is and how it will evolve in the future.

5.7.6 *Community*

The people behind the project are the driving force. Without them, a project would not exist. Therefore it is important to analyze the diverse community of each project.

COMMUNITY SIZE An important measure is the approximate size of a community. Several structural decisions and changes inside the project can depend on its size.

COMMUNICATION The communication is a vital thing in an open project. It is interesting to compare the different methods of communication in several projects depending on their development structure and size.

CONFERENCES AND MEET-UPS Meetings of developers are an important element of the development process in FOSS projects. It also shows, that there is enough interest available to provide the needed money for preparing such events.

ROLES The development process is often defined and lead by important roles in the project. Also, depending on what role a person has in the project, their influence varies.

ROLE OF THE FOUNDERS In some projects the founders are still actively involved and in some not. Depending on the project, the founders often have a very important role in the project development process.

5.7.7 *Release Process*

The release process is the action which leads to new releases. As this is a very essential part of the development processes it will be analyzed with the following classification.

VERSION NAMING Each project provides a specific version naming scheme to which they adopt and which characterizes major, minor or bug fixing releases. This information is vital to understand the project's release schedule.

CHARACTERIZATION OF RELEASES Most projects provide some kind of major and minor releases where major releases do come with new features and often backwards incompatible changes. Minor releases on the other hand often only contain fixes. It is now interesting to see where and how the line between the two is drawn.

RELEASE SCHEDULE The release schedule defines a concrete plan up to a new release and is often either time or causal dependent. As they define deadlines or important steps in the process of creating new releases it is most interesting to compare different schedules.

IMPORTANT STEPS IN THE SCHEDULE These steps often define freezes or specific points in time when all members of the project are restricted to for example not provide new code in order to increase the stability of the new release.

5.7.8 *Development*

The actual development is closely interweaved with the release process and defines how and when the development of a project takes place.

DEVELOPMENT LEAD It is hardly imaginable that projects exist which have a completely unstructured development process and have no development leaders in place. This point covers, if existent, groups of people who define new features and lead the development process.

DEVELOPMENT WORKFLOW The actual development process as often defined by the project leaders provides ways and methods to propose and develop new features for the upcoming release. It also covers the daily development and how new code finds its way into the project's repositories.

FEATURE INCLUSION PROCESS Whether projects do have established a concrete process for how and when new features are integrated in a project or not will be capped in this point.

5.8 POSTGRESQL PROJECT ANALYSIS

The PostgreSQL project facilitates an Object-Relational Database Management System (ORDBMS) which runs on all major systems such as Linux, UNIX, Microsoft Windows or Mac OS X [Pos#1; Pos#8]. The group behind the project is known as PostgreSQL Global Development Group which consists of several volunteers and employed developers. PostgreSQL is quite popular amongst its users and has won several prizes for the best database management system [Pos#2]. According to the project it is the leading FOSS database system with thousands of users and contributors [Pos#11].

5.8.1 *Project Category*

PostgreSQL is a database management system, more specifically a ORDBMS [Pos#1].

5.8.2 *Scope of Analysis*

The PostgreSQL Core Distribution will be analyzed, which consists of the PostgreSQL database server, several tools and bindings around it [Pos#6].

5.8.3 *License*

The PostgreSQL project makes use of the PostgreSQL license, which is a FOSS license [Pos#8; Pos#10]. It only requires to maintain the copyright and licensing information in the licensed source code and therefore it is quite similar to a BSD license.

5.8.4 *History*

The project was started in 1986 by Lawrence A. Rowe and Michael R. Stonebraker at the University of California in Berkeley under the name POSTGRES [Pos#9]. Not until 1996 it was developed in a university style fashion trying to explore new areas in database management systems. It was then released as FOSS by two students of Stonebraker with the new name PostgreSQL the adopted Structured Query Language (SQL). It then received a big development boost and emerged to one of the leading ORDBMSs.

5.8.5 *Community*

The PostgreSQL project has a quite large community which will be described in the following.

COMMUNITY SIZE According to the project there are six core team members, 38 major contributors and 42 contributors [Pos#4]. Adding no longer active people the number sums up to around 140. The number of total contributors might however be higher, as the project only includes people who have made contributions over a long time.

COMMUNICATION The development related communication takes mainly place through mailing lists, although also IRC channels exist. The most important mailing list for development is the *pgsql-hackers* mailing list [Pos#5].

CONFERENCES AND MEET-UPS The most important international conference is the annual *PgCon* which was first organized in 2007 [Pos#7]. However there are a lot of local conferences and workshops available.

ROLES The developers are split into three groups [Pos#4]. The core team decides on the general direction of the PostgreSQL project as well as the release cycle and the releases. Major contributors are people who introduced or maintain big features to the project. The contributors are all other who provide patches to the project.

ROLE OF THE FOUNDERS Michael R. Stonebraker and Lawrence A. Rowe did not follow the project anymore when it was published as FOSS in 1996 [Pos#9]. They are therefore no longer actively involved in the project.

5.8.6 *Release Process*

The used release process is highly structured and will be described in the following.

VERSION NAMING The PostgreSQL project uses a three digit number scheme for the releases which consists of a major, minor and micro number [Pos#14].

CHARACTERIZATION OF RELEASES The incrementation of a major or minor number defines a major release including new features and often backwards incompatible changes [Pos#14]. Such a release occurs roughly once a year [Pos#12; Pos#8]. For each major release a number of minor releases exist which are defined by incrementing the micro version number. Only bug and security fixes are allowed for those releases. Each major release is supported for five years by the PostgreSQL project. However in some cases the project can decide to drop support for a specific release if bugs cannot be resolved without risking the stability.

RELEASE SCHEDULE As major releases occur every year, the project uses the same release schedule every year along with some minor improvements from the last years [Pos#12]. The release schedule gets planned and decided during the PgCon conference. It basically starts each June with a commit review in which possible patches might be included in the next major release. Such reviews, also known as *Commit Fests*, happen four times each two months apart. In the month between an alpha release is published. After the last Commit Fest, more alpha releases can follow, if no beta releases are published. If no more critical errors are found several release candidates will be published which will lead to the next major release.

IMPORTANT STEPS IN THE SCHEDULE The Commit Fests are the only possibility to add new features to the project and are therefore a vital part of the development process [Pos#12; Pos#3].

5.8.7 *Development*

The development of the PostgreSQL project is closely entangled with the release process and will be described in the following.

DEVELOPMENT LEAD The PostgreSQL project is mostly driven by the core team [Pos#5; Pos#8; Pos#4]. Quite all decisions are made by them as well as most new features and code contributions.

DEVELOPMENT WORKFLOW Small patches are often submitted directly to the repository if they are done by a major contributor or a core team member [Pos#5]. All other patches however have to be reviewed and are therefore sent to the *pgsql-hackers* mailing list. New features are proposed in the Commit Fests.

FEATURE INCLUSION PROCESS The already mentioned Commit Fest is a periodic break in which no new development is done. However, already existing patches are reviewed and receive feedback by the core team and other developers [Pos#3]. A single Commit Fest usually runs for about one month explaining the gap of one month between each Commit Fest. One such event is lead by a Commit Fest manager who is responsible that all submitted patches will be reviewed. Often, a review results in a discussion on the mailing list and follows an acceptance, a return with feedback or a rejection. A patch can have several states, depending on whether it is in progress or finished [Pos#13]. If a patch is in progress, the following states might apply.

NEEDS REVIEW A patch is not reviewed and waiting for a review.

WAITING ON AUTHOR A new version of the patch is expected by the author.

DISCUSSING REVIEW RESULTS The review was done, however it is discussed on the mailing list.

READY FOR COMMITTER The review was done and no issues were found. It is now waiting for a final review.

The states of a patch in progress are the following.

RETURNED WITH FEEDBACK The patch was reviewed and feedback was given. A new version of the patch is expected for the next Commit Fest.

REJECTED The patch was rejected and won't make it into the next major release.

COMMITTED The patch was applied and no remaining issues are assured.

5.9 MYSQL/MARIADB PROJECT ANALYSIS

MySQL is according to the project the world's most used ORDBMS [Suno8]. It runs on all major platforms and is widely used for many web applications. MySQL was originally developed by MySQL Ab, later bought by Sun Microsystems and now owned by Oracle Corporation [Suno8; Ora10; Ora#2].

MariaDB was forked from the original MySQL when Oracle bought Sun Corporations as it was unclear how the development and licensing of MySQL would change after Oracle's acquisition [Mon#1; Mon#12]. It is intended to be a *drop-in* replacement for MySQL with full compatibility to it. Also, many of the original MySQL developers moved to the MariaDB project making it interesting to analyze both projects together.

5.9.1 *Project Category*

MySQL and MariaDB are database management systems, more specifically ORDBMSs [Mon#1].

5.9.2 *Scope of Analysis*

The analysis includes the MySQL Server package and MariaDB. Both provide a compatible database server together with several tools for running it.

5.9.3 *License*

MySQL is dual-licensed under the GNU GPL and a proprietary license. MariaDB is single licensed under the GNU GPL [Mon#10].

5.9.4 *History*

The MySQL project was started in 1994 by Michael Widenius and David Axmark by launching the company MySQL Ab [Ora#2]. Originally it was a clone of the mSQL project, which was quite popular at that time. In 2000 MySQL was released as FOSS under a dual licensing model. Sun Microsystems bought MySQL Ab in 2008 which lead to the abandonment of the MySQL founders Micheal Widenius and David Axmark [Suno8]. In 2010 Oracle Corporation bought Sun Microsystems and announced changes to the current development processes [Ora10]. As the future of MySQL in terms of FOSS was quite unclear at that time Michael Widenius forked MySQL in order to provide a community developed FOSS database [Mon#12]. The first version of MariaDB was released in 2010 as MariaDB 5.1 which is compatible to MySQL 5.1 [Mon#9]. Since then the MariaDB project tries to stay compatible with MySQL but also to provide better performance and more features.

5.9.5 *Community*

As most of the key-authors of MySQL moved to different projects such as MariaDB and since there is not much insight available about the current happenings inside Oracle, the MariaDB community will be analyzed including facts about the pre-Oracle era of MySQL.

COMMUNITY SIZE The specific size of the community is not known and is fractured due to the two acquisitions by Sun Microsystems and Oracle Corporation. The newly founded Monty Program Ab company however lists at least 20 people working on MariaDB [Mon#12].

COMMUNICATION The communication of the developers mostly takes place through mailing lists, although also IRC channels exist. The most important mailing list are the *maria-developers* and *maria-captains* mailing lists [Mon#11].

CONFERENCES AND MEET-UPS The MariaDB project has not yet set up a conference, however the MySQL project mostly meet once a year at the *MySQL Users Conference & Expo* which was first organized in 2003 [Ora#3].

ROLES The developers are split into two groups. A developer is a person who produces enhancements and new features for MariaDB [Mon#2; Mon#3; Mon#8]. However they have no right to commit their work before it was reviewed. The captains are developers who are working for a long time for the project and have made substantial improvements. To become a captain one has to make a formal request on which the other captains will vote [Mon#8]. Captains give the direction of the project, do the code reviews and take care of the project. Finally a release coordinator exists who normally is a captain [Mon#6]. By definition the coordinator gains no additional rights and only leads the communication and release management of the project.

ROLE OF THE FOUNDERS Micheal Widenius took a vital role in the fork and new project MariaDB [Mon#12; Mon#1]. He has an important voice in the community and still gives general directions, if nevertheless MariaDB is mostly driven by MariaDB captains (which he belongs too). David Axmark however left Sun in 2008 and is no longer actively involved in either the MySQL nor MariaDB project. He also does not follow their development anymore.

5.9.6 *Release Process*

The used release process is structured and will be described in the following.

VERSION NAMING The MySQL/MariaDB project uses a three digit number scheme which consists of a major, minor and micro number [Ora#1].

CHARACTERIZATION OF RELEASES The incrementation of a major number defines big and often incompatible changes to their predecessors [Ora#1]. In the current releases it marks the file format in which the database is stored. A major release is defined by an incrementation of the minor number and allows new features to be added to the release. In most cases the releases are backwards compatible, but if not, there are always upgrade paths available [Mon#9]. Finally, the micro number defines minor releases in which only bug and security fixes may apply.

RELEASE SCHEDULE The MariaDB project does not have a fixed release schedule and is quite similar to the MySQL release schedule [Mon#7; Mon#4; Mon#5]. It has criteria in place when specific releases can happen. After the last stable major release, all new features for the next stable major release will be collected. Every feature a MariaDB developer agrees to implement in the time frame to the next major release will be considered as a new feature. After most features are

nearly finished one or several alpha releases follow. The criteria for beta releases are that the proposed features are finished and no serious bugs are open. The gamma or release candidates are believed to be ready for general usage, however testing is still required. Finally the stable release should have no more open bugs or critical errors and is ready for general usage.

IMPORTANT STEPS IN THE SCHEDULE The feature proposal period, alpha and beta releases are of course the most important steps in this schedule as they restrict new features or code changes further [Mon#7]. For example, after a beta release the whole API cannot be changed anymore.

5.9.7 *Development*

The development of the MySQL/MariaDB project is loosely entangled with the release process and will be described in the following.

DEVELOPMENT LEAD The development of the MariaDB project is mostly driven by the MariaDB captains [Mon#3; Mon#11]. The founder of MariaDB and MySQL Micheal Widenius plays an important role and even if not stated by the project has an important voice about the direction.

DEVELOPMENT WORKFLOW Small patches are often submitted directly to the repository if they are done by a captain. All other patches by developers will be reviewed and applied if they suit the project well [Mon#4; Mon#5]. Bigger features will be accepted after each major release.

FEATURE INCLUSION PROCESS After a major release a developer can propose new features [Mon#5]. A feature will be accepted if a developer accepts that feature and is willing to implement it [Mon#2; Mon#3]. It then will move to the so called worklog page of MariaDB where other developers see the current status of a feature. A feature can have multiple states, such as the following.

ASSIGNED The feature was assigned to a developer who will be in charge to provide the feature for the next release.

CANCELLED The feature was cancelled for this release but might be proposed for the next.

CODE-REVIEW The feature will be reviewed by other MariaDB developers and captains.

COMPLETE The feature is ready and will be part of the next release.

IN-DOCUMENTATION The feature is completed code-wise but still has to be documented.

IN-PROGRESS A developer is currently implementing this feature.

ON-HOLD The feature will not be developed further until the status can be changed back to in progress. This could happen if a technical problem arises or the feature is put up for a discussion.

UN-ASSIGNED The feature is still unassigned and no developer has claimed this feature yet.

5.10 FEDORA PROJECT ANALYSIS

The Fedora project provides a so called Linux distribution which is a collection of FOSS and is based on the Linux kernel [Fed#15; Fed#14]. The distribution is also called Fedora operating system or Fedora project, however the latter one refers to the community which builds the project. As it only features FOSS, the Fedora operating system is also FOSS. The mission of the project is to lead the advancement and it is also known to incorporate new products and software very quickly. The project is mostly sponsored by Red Hat, however due to its structure it can be seen as independent from a companies influence.

5.10.1 *Project Category*

The Fedora operating system is, as the name already states, an operating system [Fed#15].

5.10.2 *Scope of Analysis*

The operating system provided by the Fedora project will be analyzed, which consists of a large collection of FOSS needed to run and work on a computer.

5.10.3 *License*

As the projects goal is to provide a FOSS based product only FOSS licenses are allowed [Fed#10]. The project provides a list of acceptable licenses. For own software creations the GNU GPL is mostly used.

5.10.4 *History*

The original Fedora project was created in 2002 by Warren Togami in order to enhance the quality and number of packages available for

the at that time existing Red Hat Linux distribution [Fed#15; Fed#14; Fed#8]. In 2003 the development of Red Hat Linux stopped and was merged with the Fedora project. Since then the Fedora project provides a community distribution while Red Hat Enterprise Linux is an officially supported Linux distribution which derives from Fedora versions. Until 2006 the operating system was known as Fedora Core. Later the project changed the name to Fedora. The name Fedora originates from the Red Hat logo which shows a person with a fedora hat.

5.10.5 *Community*

The Fedora project has a quite large community which will be described in the following.

COMMUNITY SIZE The exact size of the community is unknown, however the project consists of a quite large group [Fed#13]. For example the total number of active accounts exceeds 30,000 people. This number however does not describe the actual size accurately as it includes many users of the operating system.

COMMUNICATION Most of the communication takes place over mailing lists [Fed#15; Fed#9; Fed#12]. Each of the mailing lists is quite focused on a certain sub-topic, however the most important general development related mailing list is the *devel* mailing list. Additional communication happens through several IRC channels, forums, blogs or the weekly newsletter [Fed#6; Fed#2].

CONFERENCES AND MEET-UPS Since 2005 the Fedora project organizes the Fedora Users and Developers Conference (FUDCon) which is held annually in various places around the world [Fed#7].

ROLES The Fedora project has a quite flat structure with several teams and Special Interest Groups (SIGs) [Fed#9; Fed#2; Fed#12]. Each team or SIG is in charge for a specific sub project or area in the project. Each team is lead by one or several people. The usual way to be a part of a specific team is to provide several contributions until one can apply. Examples for such teams and SIGs are Release Engineering, Desktop or Usability. The technical leadership is handled by the Fedora Engineering Steering Committee (FESCo), a community elected team, which handles new features, SIGs or technical matters [Fed#4]. The general direction and guiding decisions however are handled by the Fedora project board which consists of five community elected people and four appointed by Red Hat [Fed#1].

ROLE OF THE FOUNDERS Warren Togami is still involved in the Fedora project, however he focuses more on Fedora related projects, such as SpamAssassin or K12Linux which is a distribution built on top of Fedora [Fed#14].

5.10.6 *Release Process*

The release process used is highly structured and will be described in the following.

VERSION NAMING The Fedora project uses a single number versioning scheme [Fed#8; Fed#5]. This single number defines major releases.

CHARACTERIZATION OF RELEASES There are no minor releases since updates to the operating system come via single updates of the affected packages [Fed#8; Fed#5]. Therefore each incrementation of the version number defines a new major release.

RELEASE SCHEDULE The Fedora project uses a fixed release cycle with a new major release every six months [Fed#5; Fed#11]. Each major release is then maintained until one month after either two releases or 13 months. The release schedule is proposed by the release engineering team and approved by the FESCo. In some cases, the schedule can be slightly adjusted if critical bugs appear and can't be solved in the original time frame. After a major release the planning and development can start. One week before the alpha release, no new feature will be accepted. The alpha release will be followed by a beta and final release candidate. Depending on whether each of the previous releases was postponed or not the final release will be published approximately six months after the previous major release.

IMPORTANT STEPS IN THE SCHEDULE The most important steps are the feature acceptance, feature freeze and feature complete milestones in the release schedule [Fed#5]. In each of the named steps the FESCo will decide if a feature will be accepted or be in the next release depending on its completeness.

5.10.7 *Development*

The development of the Fedora project is quite entangled with the release process and will be described in the following.

DEVELOPMENT LEAD The development of the Fedora project is mostly driven by the FESCo which decides on new features and the technical direction of the project [Fed#4].

DEVELOPMENT WORKFLOW The development process of the Fedora project is quite open [Fed#11; Fed#12]. As already stated above, there are many SIGs available which steer the development of a certain area in the project. Depending on the SIG the development workflow can vary, however the general direction is always given by the FESCo.

FEATURE INCLUSION PROCESS To propose a feature for the next major release a formal feature proposal has to be made [Fed#3; Fed#4]. This includes a description of the problem, an owner, the current status and several other points. Any Fedora community member can propose them and the FESCo decides on the acceptance of the feature. Additionally a feature can be dropped if it is not completed at the feature freezer, before the beta release or if the owner does not update the feature status. A feature can be either incomplete when the proposal is not ready, ready when it is disposed for review, ready for FESCo when it can be proposed to the committee and finally accepted if the voting by the FESCo was successful.

5.11 DEBIAN PROJECT ANALYSIS

The Debian project provides an operating system on top of various kernels, such as the Linux kernel [Deb#2; Deb#13]. Even if there are other kernels available, such as the FreeBSD, NetBSD or Hurd kernel, Linux is the most prominent one. Therefore, Debian often gets called Debian GNU/Linux. The mission of the Debian project is to provide a free operating system in the meaning of FOSS, to provide a full featured operating system with high standard and to have a big evolving community. The Debian project is known as a very stable operating system and is therefore often used on servers.

5.11.1 *Project Category*

The Debian operating system is, as the name already states, an operating system.

5.11.2 *Scope of Analysis*

The Debian GNU/Linux operating system provided by the Debian project will be analyzed, which consists of a large collection of FOSS needed to run and work on a computer.

5.11.3 *License*

The project's goal is to provide an operating system consisting entirely of FOSS [Deb#11; Deb#14]. However, it is possible to install non-

free software too. According to the Debian Free Software Guidelines (DFSG) the project itself only accepts FOSS licenses, but the GNU GPL is mostly used for own creations.

5.11.4 *History*

The Debian project was created by Ian Murdock in 1993, in order to provide a distribution which was developed in an open style and in the spirit of a FOSS development process [Deb#2; Deb#1; SSRD08]. At that time, the concept of a distribution was quite new and Debian can be counted as one of the early distributions, although it was not the first. The name is a fusion between Ian and the first name of his wife, Debra. The Debian project established several policies and contracts to ensure the future freedom of the project. Debian is still the most significant distribution that is not backed by a commercial entity.

5.11.5 *Community*

The Debian project has a large community available which will be described in the following.

COMMUNITY SIZE The community size is quite large and ranges between 800 and 1000 active developers [Per11; Deb#10].

COMMUNICATION Most of the communication is handled through several mailing lists [Deb#12; Deb#14; Deb#6]. For quite every aspect or subproject there is a mailing list available. The most important for development is the *debian-devel* mailing list. Additionally there are many IRC channels, forums and blogs available.

CONFERENCES AND MEET-UPS The most important conference in the Debian project is the annual *DebConf* which took first place in 2000 [Deb#3]. So far it was organized in various regions and continents worldwide.

ROLES The Debian project has a quite hierarchical structure with several teams [Deb#10; SSRD08]. A Debian contributor is any person who contributes to the project, but has no additional rights [Deb#14]. Debian maintainer have restricted access capabilities [Deb#5]. To become a Debian maintainer one has to go through a formal process requesting the role. At least one Debian developer must then approve this person. The Debian developers have full access rights and often maintain parts of the project [Deb#4]. On top there is a project leader who gets elected by the Debian community every year [Deb#10; Deb#9]. Beneath the project leader there is a technical committee which drives the project's technical decisions. Additionally

there are many teams in charge for parts of the project, such as Release Engineering, Ports or the distribution of the project.

ROLE OF THE FOUNDERS Ian Murdock led the project from its beginning until 1996 as the project leader when he passed this function to Bruce Perens [Deb#14; Deb#1]. Although he still works in the FOSS field he is no longer involved in the Debian project.

5.11.6 *Release Process*

The used release process is highly structured and will be described in the following.

VERSION NAMING The Debian project uses a three digit number scheme which consists of a major, minor and micro number [Deb#8].

CHARACTERIZATION OF RELEASES Debian has always at least three active releases available: stable, testing and unstable [Deb#8; Deb#7]. Each of the three is a major release and indicated by a different major version number. The minor number was used until Debian 3.1 as a major release indicator, however this is not the case anymore. The micro version number indicates bug fixes to the major release.

RELEASE SCHEDULE With three active release branches, the actual releases change if a new stable version will be introduced [McG11; Deb#7; Deb09]. In that case the old stable will be named as oldstable and maintained for one more year. The testing branch will become the new stable and the unstable the new testing. The Debian project does not have a fixed release schedule. A new release is announced when the release team and core team think it is ready. At this time a freeze is called in and a release follows shortly thereafter. However there is some effort to have regular releases and predictable freeze schedules. For the next release the Debian project is trying to adopt a time based freeze and release cycle [McG11].

IMPORTANT STEPS IN THE SCHEDULE As the new release schedule is not clear yet, a characterization of the important steps is not possible. However the final freeze and the announcement of release goals by the release team are certainly important steps within the schedule.

5.11.7 *Development*

The development of the Debian project leads directly to future releases and will be described in the following.

DEVELOPMENT LEAD The development is mostly driven by the project leader and the technical committee who set the goals for future releases [Deb#10]. In addition the team leaders can set their own goals.

DEVELOPMENT WORKFLOW As the stable branch is closed for new features, most new features have to go directly into the unstable branch [Deb#14; Deb#4; Deb#7]. If those changes have been tested for long enough they eventually will be merged with the testing branch. Parts of the Debian project will either be maintained by single Debian developers or co-maintained by others. Depending on that and the goals set by the project leader, Debian developers start to work.

FEATURE INCLUSION PROCESS The Debian project uses so called release goals which are proposed by Debian community members and chosen by the release team [Deb09; McG11]. The release team will decide on each goal and set it for a specific release or postpone it. At some point during the development the release team will announce the chosen release goals.

6 COMPARISON OF DEVELOPMENT PROCESSES

As the analyzed projects were described in the previous chapter, this chapter will compare the development processes and project details with each other.

6.1 PROJECT ORIGIN

All analyzed projects share a quite different history. Some were developed with an university background like PostgreSQL or Python while other projects were born in the FOSS context. The GNOME or Debian projects are an example for that. Projects like Plone or Fedora were an improvement and additional layer to an existing project.

It is interesting to note, that many project founders tried to solve a personal problem with the project they were about to start. KDE, Drupal, Fedora and PHP are notable here. However also commercial or philosophical background can initiate a project, such as MySQL with MySQL Ab or GNOME show.

PROJECT	LICENSE
Debian	Various, mainly GNU GPL
Drupal	GNU GPL
Fedora	Various, mainly GNU GPL
GNOME	GNU GPL, GNU LGPL
KDE	GNU GPL, GNU LGPL
MySQL/MariaDB	GNU GPL
PHP	PHP License
Plone	GNU GPL
PostgreSQL	PostgreSQL License
Python	Python Software Foundation License

Table 6.1: Used licenses in the analyzed projects.

Even if the number of developers increased over time, all projects were started by a small group of people which often consisted of at most two people. Often, a single person was more involved in the project creation, even if other people were involved too. Michael Widenius of MySQL/MariaDB fits this case. After inception however the number of people involved grows rapidly. This statement holds for all analyzed projects and is especially the case when the project was

PROJECT	ORIGIN	FOUNDER
Debian	1993	Ian Murdock
Drupal	2001	Dries Buytaert
Fedora	2002	Warren Togami
GNOME	1997	Miguel de Icaza & Federico Mena Quintero
KDE	1996	Matthias Ettrich
MySQL/MariaDB	1997	Michael Widenius & David Axmark
PHP	1994	Rasmus Lerdorf
Plone	1999	Alexander Limi & Alan Runyan
PostgreSQL	1986	Lawrence A. Rowe & Michael R. Stonebraker
Python	1989	Guido van Rossum

Table 6.2: Project founders and origins.

finally released as FOSS. Examples like PostgreSQL or Python show the rapid increase of people involved, be it just developers or other, when finally released as FOSS. The growth graph however is quite similar in all analyzed projects, if one considers time and size differences between the projects. Also it is worth noticing that almost every analyzed project has one or more rapid developer growth periods. Those periods can be matched with a certain major release of the projects. This becomes clear when one examines the release and development of Python 2.0, GNOME 3.0, KDE 4.0 and Plone 4.0. All of those versions are major releases with new features and approaches to their project goals. It seems that the development of such big releases attracts a lot of new developers. Furthermore the increased size stayed almost the same after the release.

6.2 COMMUNITY

The previous project analysis has shown, that the communities of each project seem to be quite different. This will be compared in the following sections.

6.2.1 *Community Size*

As all of the analyzed projects share a quite long history, a lot of people were active in the projects. This of course biases the comparison between communities, as the actual number of developers is almost always lower than the total sum of developers in the project history. Also taking in mind that only the central parts of the projects were an-

alyzed, the resulting number can vary a lot. For example the GNOME project claims that over 3500 people contributed to the project. The fact that only GNOME Core was analyzed and no longer active developers were left out reduces the finding to currently about 350 active developers of GNOME Core. The same probably holds true for the Drupal project with about 1200 enlisted developers and for the KDE project with about 1800 active contributors.

Also it is important to carefully distinguish between core developers and other contributors. Since in some projects only core developers have access to the code repository, an analysis of code authors is not always a satisfying approach. The Drupal project for example only has about six to ten distinct authors to Drupal Core. Only recently they adopted to give credit to the original author instead of referring to them in the commit message. Another interesting insight is the Plone project which claims to have about 300 active core developers, the code repository however only shows a number between 50 to 70 active developers at the same time. The PostgreSQL project on the other hand gives an exact number of active and retired core developers, only leaving out the number of minor contributors.

It is not always the case that the actual number of developers is smaller. Projects like Fedora or Debian with a broad scope and smaller tasks like package creation can actually have a number of active developers which matches the claimed number. However one has to keep in mind that the comparison of a core piece of a project with a full operating system isn't valid and has low relevance. That said the size comparison must be analyzed carefully with the above considerations in mind.

6.2.2 *Communication*

On a worldwide distributed project the communication within a project is one of the most vital parts. Every project has a wide array of communication methods in place. Quite every project uses mailing lists as its favourite communication method. A plausible explanation for this is probably the fact that it is easily scalable, open and archived.

Although every project has lots of different lists for quite every aspect within the project, there is always one mailing list which is the most important. This is true for GNOME with the *desktop-devel*, PHP with the *internals*, Plone with the *plone-developers*, Python with the *python-dev*, PostgreSQL with the *pgsql-hackers*, Fedora with the *devel* and Debian with the *debian-devel* mailing list. In some projects this single mailing list is split into two, one more general list for the project and the second for the core project. This is the case for KDE with the *kde-devel* and *kde-core-devel* mailing lists and MariaDB with the *maria-developers* and *maria-captains* mailing lists. An exception is the

Drupal project which uses mailing lists however the most important communication channel is a mailing list akin forum on their project website.

Additionally in every project there are lots of other direct and indirect communications such as IRC channels, blogs, newsletters and others.

6.2.3 *Conferences*

As most of the developers are living in different cities, countries or continents, the communication is almost always handled through the internet. Therefore conferences and meet-ups provide the possibility to meet in person, plan and design future goals more easily and to invigorate the community. Each analyzed project organizes one or more annual conferences in place where developers and users can meet. It is interesting to note that only after years communities started to organize them on a regular basis. It also seems that most projects existed since years before the first conference was held. Along with the growth of a project the number of attendees at a conference grew as well. Another interesting fact is that some projects include the conference in their release and development schedule, for example the GNOME project in which the conference provides an extra development boost for the upcoming release in autumn.

Due to the size of the project and the worldwide distribution of developers, each project organizes in addition frequent local meet-ups, hackfests, sprints, smaller conferences or other workshops. These however focus more on specific parts or goals of the project.

6.2.4 *Roles*

The analysis of the projects has shown, that all projects use a fairly hierarchical development structure. Starting from the bottom of the hierarchy there is always room for casual developers or contributors. Moving one step up, developers or contributors get more rights but also more responsibilities. For example one manages certain parts of the project or provides guidance to new developers. Above these maintainers of a certain subsystem is always a rather small group managing the project's development. Then, at the top there is a project leader or a group filling the project leader role. It is also interesting that some projects split the same function into two roles. Keeping this fact and the different project scopes in mind, the used structures are middlingly similar, especially if the projects are working within the same field.

Of course this structure varies over time and size of the project. For example the Debian project which has a very long history in comparison to others has a filed segmentation with clear duties and respon-

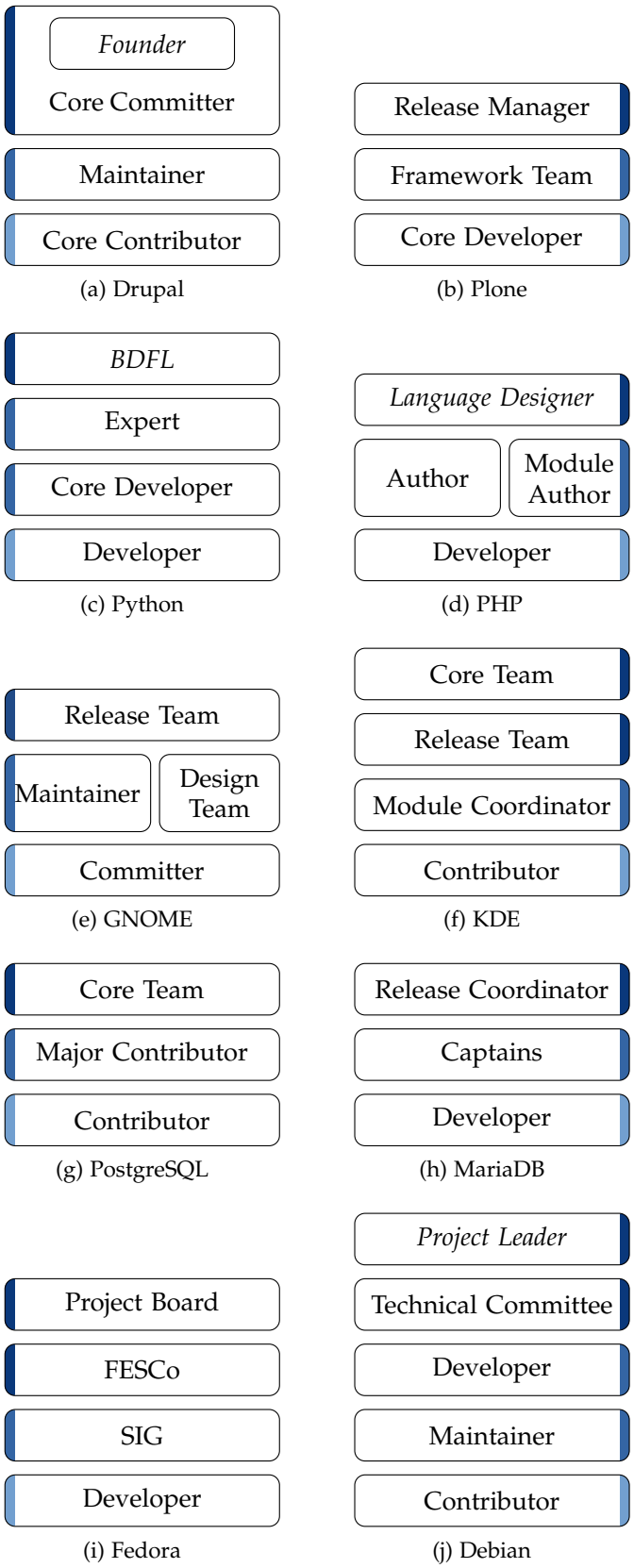


Figure 6.1: Comparison of development related roles in the projects. Project leaders are printed in *italics*, founders, leaders and release coordinators are marked with ■, core developers are marked with ■ and co-developers are marked with ■.

sibilities for each role. A younger project like MariaDB has a simpler model, which nonetheless has the described groups on top.

A release team is a critical part of a project, either as a dedicated team or represented by the project leader. Furthermore the concept of having people who are responsible for certain subsystems seems to be widely established across all bigger projects.

It is also interesting to note, that in projects where the project founder is still active, the structure is more hierarchical with the founder on the top. This is true for Drupal, PHP and Python. Debian is a special case because the original project leader passed on his duty, but preserved the project leader hierarchy. Sometimes the leader gathers other people around him trying to be not solely responsible for the project, such as in the MariaDB project. In projects where the leader is no longer involved or does not hold any power, project leader teams were established. This is true for KDE, Fedora and GNOME.

Even if projects claim to be friendly to newcomers and an open community, the structures analyzed here show, that one always has to start at the bottom of the hierarchy, earning trust and providing enhancements to the projects before being promoted on holding a carrying business.

6.2.5 *Project Founders*

The project founders played a vital role during the inception of the project. It was already explained that even if there are more project founders, often one single person plays a leadership role. It was also pointed out that some project founders are still active and lead the project, while others have left their project.

Therefore the project founders of the analyzed projects can be classified into three categories. First, project founders which are still active in the project and have a leadership position. This is the case for the Drupal project, as well as for Python and PHP. In all of those projects, the development structure might look different without the leader. They define goals, releases, new features and are often the decision maker. The opposite are projects where the original founder no longer is present. Fedora, Debian and PostgreSQL are good examples here. All named projects established a leadership group which defines the future of the project. Debian still has a project leader, who gets voted by the community. At last there are projects, where the original leader is still active and sometimes gives input but has no power. In the KDE and GNOME projects the founders are working on related projects or topics and sometimes give their opinion, but they can't decide anything. In the Plone and MariaDB project the founders are members of the leadership team but have no special role. At most

they can give their opinion and community members might give their notion a higher value.

6.3 RELEASE PROCESS

The release process seems to be the most diverse item in this analysis, as most projects have gone through different processes and workflows during their time of existence. Nevertheless some of the different stages of the projects seem to be quite equivalent to each other.

6.3.1 *Versioning Scheme*

All projects, except Drupal and Fedora, use a three digit version naming scheme. It consists of a major, minor and micro number, which characterizes a release. The Drupal project uses a two digit versioning scheme, which defines major and minor releases. It is important to note, that the minor releases of a major release only include bug fixes and therefore are following the scheme of a major and micro version number. The Fedora Project just uses a single number to represent major releases. Due to the structure and automatic update process, bug fixes can be included automatically in a running system. For comparison the Debian project also pushes bug fixes to the users, however it additionally releases those changes as further bug fix releases.

It is true for all projects, that an increment in the major number defines often a backwards incompatible release with new features. Such a major release is often a ground breaking change and often defines a new era for the project. A major releases is generally extremely scarce. This is also reflected by the comparatively low major number the projects have, for example GNOME 3, KDE 4, Plone 4, Drupal 7 and so on. Fedora is an exception here, as every new release increments the major number by one, which currently is 16.

The minor number almost always defines a feature release which is backwards compatible to the previous major release. There are some exceptions where this is not the case, for example in the PHP 5.x release cycle. However those changes can be seen as minor when comparing them to the incompatibilities of two major releases. Also, if such changes occur, most projects provide an upgrade path. Except Drupal and Fedora, all projects use such a number.

In some projects, an increment of the minor version number defines a major release. In other words, regardless of whether the major or minor number changes the release is always a major release. This is true for GNOME, PHP and Debian.

The GNOME project always increments its minor number by two to define a stable release cycle. An odd minor number therefore always defines development releases.

PROJECT	MAJOR	MINOR	MICRO
Debian	✓	✓	✓
Drupal	✓		✓
Fedora	✓		
GNOME	✓	✓	✓
KDE	✓	✓	✓
MySQL/MariaDB	✓	✓	✓
PHP	✓	✓	✓
Plone	✓	✓	✓
PostgreSQL	✓	✓	✓
Python	✓	✓	✓

Table 6.3: Used versioning schemes in the analyzed projects.

Micro releases are a number of subset releases for the above mentioned feature releases. They are only allowed to include bug and security fixes, no new features and no incompatibilities. Except Fedora, all projects use such a number.

In a nutshell it can be said that all projects use a similar versioning scheme with some minor exceptions, which do not have an all to big impact.

6.3.2 *Release Schedule*

The release schedule leads to new major releases and often includes minor and bug fix releases. They are therefore a vital point for each project's future. As such most projects have established a rather detailed release schedule which they accurately try to follow. Nevertheless there are some differences while comparing each project's schedule which will be outlined in the following.

The most distinguished difference is the decision of a fixed release cycle. A fixed release cycle iterates over a given time frame and provides expectable release dates. Projects like Plone, GNOME and KDE projects use a fixed release cycle with a duration of six months. It means, that each year there will be two major releases including a number of minor or bug fix releases. Also Fedora uses a similar cycle, as it is more or less bound to the GNOME release cycle. The PostgreSQL and PHP projects use a fixed release cycle with a duration of one year.

Although projects like Drupal and Python do not use a fixed release cycle for major releases, they do however use a fixed release cycle for minor and bug fix releases. The Python project publishes minor releases roughly every 18 months and bug fix releases roughly

every six months. The Drupal project publishes bug fix releases on a monthly basis.

Only MySQL/MariaDB and Debian do not have a fixed release cycle. At least the Debian project is currently trying to provide a fixed release schedule.

All projects however have some sort of freezes as part of the schedule, such as code, features, translatable strings, documentation and more. Freezes should help to confine areas where new bugs or drawbacks could appear. Of course they are restricted to new development and developers can still apply bug fixes. However they will be analyzed deeply before they can be applied. As such freezes are a vital part of all analyzed project schedules. Also, they often keep in effect for the released major branch. One of the most obvious consequences is the fact that bug fix releases only include fixes and no new features or user interface changes.

The most interesting facts are when the freezes occur and with which restrictions. The GNOME project as well as the Debian project have established a freeze shortly before their final release including code, feature, user interfaces, API and string freezes. The freeze period of the GNOME project however has a defined start and end date, since the release schedule is a fixed one. The KDE project on the other hand distributed the different freezes more evenly throughout the release cycle. It also differentiates between soft and hard freezes. The freezes are set in order and always have a soft freeze before a hard freeze. This applies for features, strings and API freezes. The Plone project is set somewhat in the middle having a feature freeze two months before the final release and then applying more and more restrictions during the releases of alpha, beta and release candidate versions. The Fedora project is quite similar to that approach, however has its feature freeze roughly three months before the final release. Following an alpha and beta release more restrictions apply.

The Python project only allows new features until the first beta release which occurs about two months before the final release. After that point only bug fixes can be applied.

The PHP project has changed the layout of its schedule a lot during the last years while trying to elaborate its yearly release schedule. Currently the feature freeze is in act about eight months before the final release, confining other freezes more and more until the first alpha release. Further beta releases from that point until the final releases do only get bug fixes.

The flexible release schedule of MySQL/MariaDB forces the project to establish criteria when specific releases can happen. A such criteria for a feature freeze is the publication of alpha releases. After that no new features are allowed and only bugs get fixed. This continues until the final release is considered to be bug free.

(a) Drupal (b) Plone

(c) Python (d) PHP

(e) GNOME (f) KDE

(g) PostgreSQL (h) MariaDB

(i) Fedora (j) Debian

Figure 6.2: Schematic representation of major and minor development cycles of the analyzed projects. Feature inclusion phases are marked with ■ ■ ■, freezes are marked with ■ ■ ■ and releases are marked with ■ ■ ■.

Like MySQL/MariaDB the Drupal project does not have a fixed release cycle and therefore arranges the development and freezes into different phases. Each phase comes with its freezes in the following order code, API, feature, user interface and string freeze. After all freezes have been declared a number of alpha, beta, release candidate and lastly the final release is published.

The PostgreSQL has replaced a single feature freeze with a series of Commit Fest which were explained earlier. In fact it is the only analyzed project which does not have a single freeze but a series of development milestones which limit new features and development time by time.

It is also interesting to note, that every project uses some kind of alpha, beta and release candidate versions before publishing the final release. The naming is of course different. For example the GNOME project does not provide a special naming, except for the odd minor number, while other projects, such as Drupal or Debian name their release accordingly.

6.4 DEVELOPMENT

While the release cycle defines the process around new releases, the development process of a project defines the actual shape of it. As such it is vital to understand how projects are being lead, structured and divided.

6.4.1 *Development Lead*

The development of a project is always lead by a single person or a group of people. All analyzed projects' development had a form of leadership. Each project showed a hierarchical structure when it came to decisions related to the development of a project.

Often the development is lead by the founder, if they are still active. In the Python, Drupal and PHP projects the original founder has a great influence when it comes to decide on changes or new features. Projects like Fedora, MariaDB, Plone and PostgreSQL have established leadership groups who define the direction of the project as well as decide over new features. While the Fedora and Plone project have an arbitrary or elected group they do not necessarily provide enhancements themselves. The MariaDB and PostgreSQL projects on the other hand are mostly enhanced and lead by the same group.

While the KDE and GNOME projects have established core or release teams they do not necessarily have power on areas other than their duties. Also due to their modular structure, the concrete development is mostly lead by the maintainers of single areas. A minor exception is GNOME's design team which is involved in the shap-

ing of user interfaces. Since the team cannot dictate other GNOME members what to do, the above assumption still holds.

The Debian project is structured very hierarchically and as such the overall direction is set by the elected project leader. The technical committee then looks after the concrete implementation while the single team leaders are in place for certain areas of the project.

6.4.2 *Development Workflow*

The development workflow of a project defines how changes and enhancements are brought into the project. Some of the analyzed projects have established a feature inclusion process which will be highlighted in the next section.

In all analyzed projects it is possible to commit small patches and enhancements directly to the project's repository or have it committed by a core contributor if one has not enough rights. Such changes are not tied to an existent feature inclusion process. As such this often only comprises bug fixes and other changes similar in size. This is often assisted by a bug tracking system which each of the analyzed project uses.

It seems that all changes committed to the repositories are reviewed by other members of the project. In case of not gaining acceptance the patch is often discussed in the project's mailing lists or other suitable means of communication. It is also possible that patches are sent to the mailing list or to a bug tracking system and discussed there before being applied. Quite all projects have some sort of team leaders or maintainers who can decide about a certain patch for a certain subsystem of the project.

6.4.3 *Feature Inclusion Process*

The feature inclusion process defines, if established, how and when features or big changes come into a project. In the analyzed projects a quite diverse passel of processes has been found. It ranges from a very structured and well documented process like the one in the Python project to a more dynamic approach like the one in the KDE project.

The most structured and well laid-out process is Python's PEP process. It defines a well documented approach of new features and big changes from its inception until either acceptance or withdrawal. This process seems to work well for the Python community as well for development, community and leadership processes.

Interestingly other projects started to adopt this process with minor changes. For example the PLIP process, which is used by the Plone project only uses it for development enhancements and not for com-

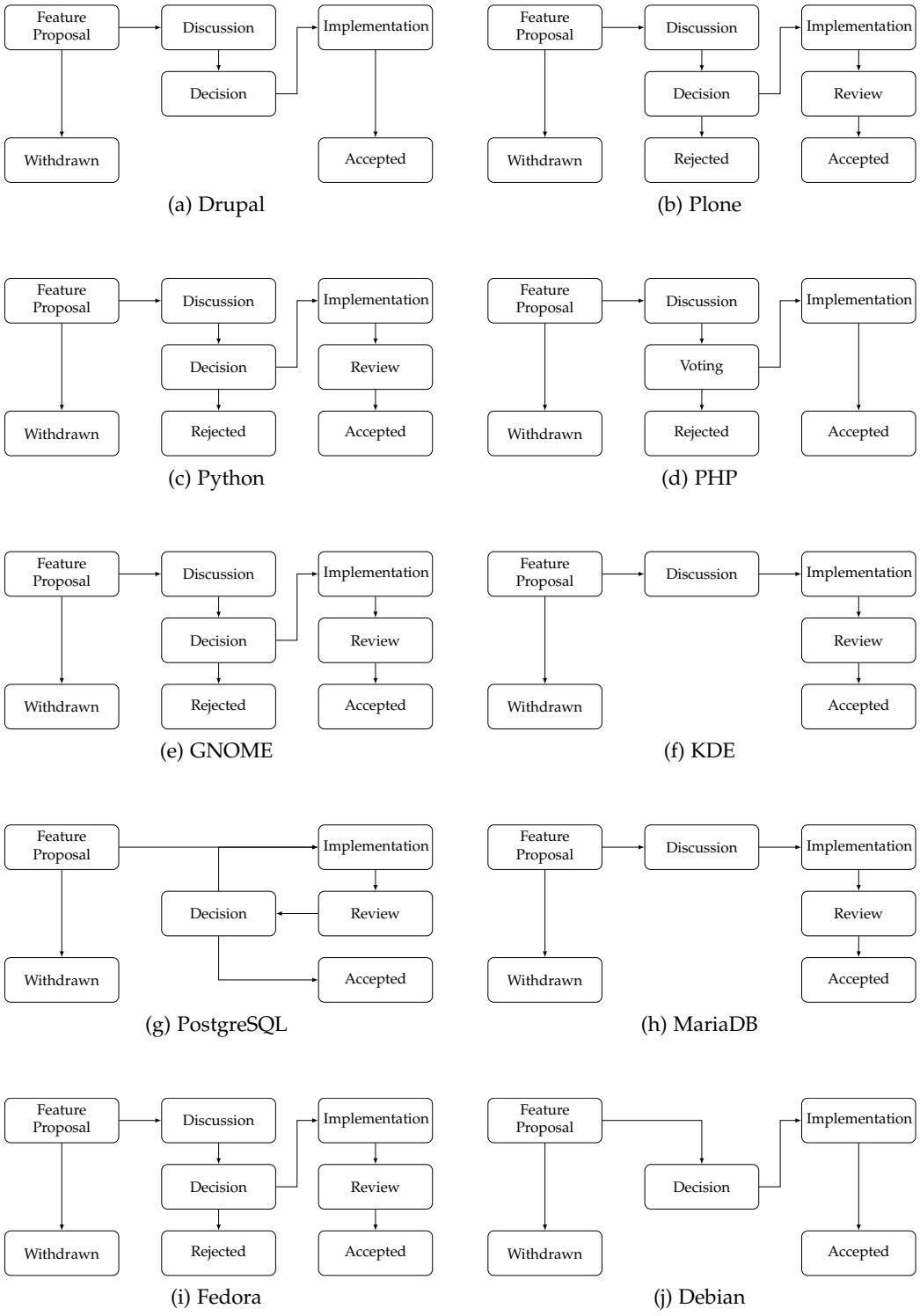


Figure 6.3: Schematic and simplified representation of feature inclusion processes of the analyzed projects.

munity issues. As such the process is a simpler and more focused to development compared to the PEP process.

Also the PHP project adopted a version of the PEP process which is called RFC. Since it also focuses on development, it is similar to the PLIP process. A minor difference is the separate voting step on each proposal, which is not existent in the other two.

A slightly more dynamic approach is the Drupal initiative process where each Drupal developer can propose an initiative with a certain goal. At a certain point in time the Drupal project founder and leader Dries Buytaert chooses the initiatives the project should follow for the next release.

Similar to this, but slightly more structured and not backed by a single person is the feature based development process of GNOME and Fedora. In both projects, features can be proposed which are then chosen by the GNOME release team and the FESCo. In both projects proposals follow a certain format and are more structured when compared to Drupal initiatives. Also, in both projects the named groups have the possibility to drop or postpone a certain feature if it is not ready for the upcoming release.

The Debian project uses goals which can be proposed by community members and are chosen by the release team. This is antithetic to the previous projects, since goals are just single wordings without any structure. No concrete steps, involved parties, code or other are trailed to the goal.

The KDE and MariaDB projects use an even more dynamic approach where members can propose features, but they have to be picked up and implemented by developers. As such there is no real instance which could prohibit the development of a certain feature.

The PostgreSQL project is the only project which has removed such a specific inclusion process altogether and replaced it with a series of Commit Fests. A feature can be brought into a Commit Fest and evolves over time until it can be committed. Given the analysis such a series of feature reviews is equivalent to a series of feature inclusion processes where the goal is always to get a certain enhancement into a project instead of discarding it directly.

7 DISCUSSION

With the analysis performed and the single findings compared, this chapter offers a discussion and comparison with related findings and the presented software engineering methods. Finally the main research results are outlined.

The probably most obvious common ground in the analysis was the establishment of a project analysis catalogue, with which FOSS projects could be analyzed systematically and satisfactorily for the goals of this thesis. While the previous chapter described the differences between the projects, they still share general mutuality of structure and processes. So to say the projects are similar with equal structures and processes when viewed in a simplified manner.

Another important issue to note is that although it was not examined in this analysis, all projects seem to evolve their structure and processes over time. This was shown by Scacchi [Sca06], Godfrey and Tu [GT00] but also by Johnson [Joh01] who claims that all FOSS projects start as a closed prototype and evolve into an open project over time.

7.1 PROJECT ORIGIN

As shown before, most projects are initiated to solve a personal issue or problem. This fact has also been analyzed in related studies, such as Raymond [Ray98], Lakhani and Wolf [LW03], Hertel, Niedner and Herrmann [HNH03] or Johnson [Joh01]. In some cases the projects develop from existing projects or scientific studies, such as the Python project. In other the project is launched by a company with the primary motivation of economic success.

The analysis has shown, that despite the age of a project, many still grow at least linearly. Some projects seem to maintain stability over time, while others, such as the PHP project, seem to decline. Similar results have been shown by Godfrey and Tu [GT00], Roets, Minnaar and Wright [RMW07] or Ogawa et al. [Oga+07]. Schweik and Semenov [SS03] identified a life cycle with three phases: project initiation, going open and project growth, stability or decline. This definition lines up very well with the findings of this analysis.

7.2 COMMUNITY

In contrast to the Bazaar model by Raymond [Ray98] the project structure in the analyzed projects was always hierarchical, even if

there was only a flat hierarchical structure. This affirms the findings by Crowston et al. [Cro+05]. Also Bezroukov [Bez99#1; Bez99#2] criticizes the Bazaar model as too simplistic and not matching. As noted earlier, Ghosh [Gho05] analyzed several different models ranging from hierarchical to completely flat, Bazaar like structures. The latter was not identified in any of the analyzed projects and even if they did differ in terms of hierarchy levels, the general hierarchical structure was similar. That of course only is proven for the analyzed projects and not for all FOSS projects.

This repeats itself when one analyzes the project structure with a a leader or leadership group who drive the project. This correlates with the findings of Johnson [Joh01], Crowston and Howison [CH05], Warsta and Abrahamsson [WA03] and Krishnamurthy [Kri02].

While the analyzed projects do have a hierarchical structure, the communities are nevertheless very welcoming and it seems to be relatively easy to enter a community and step up the ladder. In this aspect the analyzed models resemble the Bazaar model, which claims that it is very easy for new developers to join the project and take charge of important parts of the project.

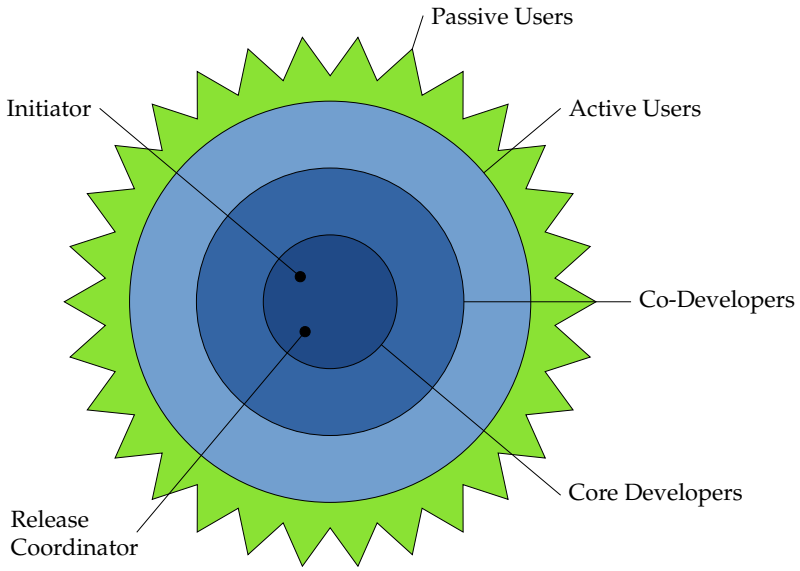


Figure 7.1: Free and Open Source Software project development structure as proposed by Crowston et al. [Cro+05].

Regarding the communication within the projects, the mailing list is the most used communication method in FOSS projects as shown by Schweik and Semenov [SS03], Ogawa et al. [Oga+07] and Kim [Kim03]. This finding was considered to be true, as all projects do have mailing lists and in fact the most important communication channel seems to be the mailing list. Most projects do have several mailing lists in place with one or two being the most important ones where the development of a project is planned.

7.3 RELEASE PROCESS

An interesting fact is the move towards fixed release cycles. Most of the analyzed projects adapted a fixed cycle and others are in the process of transition. Fixed cycles can also be found in agile software engineering methods, such as Extreme Programming or the Scrum method. While not equal to those methods, the analyzed projects seem to benefit from such an approach. The release schedule with fixed cycles is often already fixated before development starts and each project tries to stick to it as close as possible. Development releases and freezes are quite similar in all projects. The only main difference is the number of development releases and freezes and the point in time when freezes occur and what part of the project they do cover.

As shown by Mockus, Fielding and Herbsleb [MFH02] the role of the release manager is vital for every project. The analysis reflects this since each project has either a single person or a team in place to manage releases, freezes and all other related duties. Also Crowston et al. [Cro+05] rates the release managers as one of the most important people next to the founder or leader. In some cases, this role is also the same, especially in new projects.

7.4 DEVELOPMENT

While it certainly is not used in mature projects, the Waterfall model by Royce [Roy70] often appears in new projects or prototypes. It is also interesting to note, that the phases described in the Waterfall model are often found in development processes of FOSS projects as Roets, Minnaar and Wright [RMW07] point out. Iterative or evolutionary software engineering models however are the most common pattern, especially in mature projects. The Spiral model by Boehm [Boe88] is a good example and gives a good first match when compared to the iterative releases and development schedules. Roets, Minnaar and Wright [RMW07] however make a strong case for the Spiral model not being used or found in FOSS projects.

Agile methods seem to be much more appropriate when comparing them with FOSS development processes. The iterative process, the close relationship with the user and quick development and prototyping seem to make an argument for agile methods. However Koch [Koc04], Warsta and Abrahamsson [WA03] proof differences between agile methods and FOSS development methods. They point out an array of differences, most importantly the non available co-location of developers, in which the processes differ. However they agree that the processes show clear similarities. For example the development workflow of the PostgreSQL with its regular Commit Fests resembles important steps in the Scrum software engineering method, in the

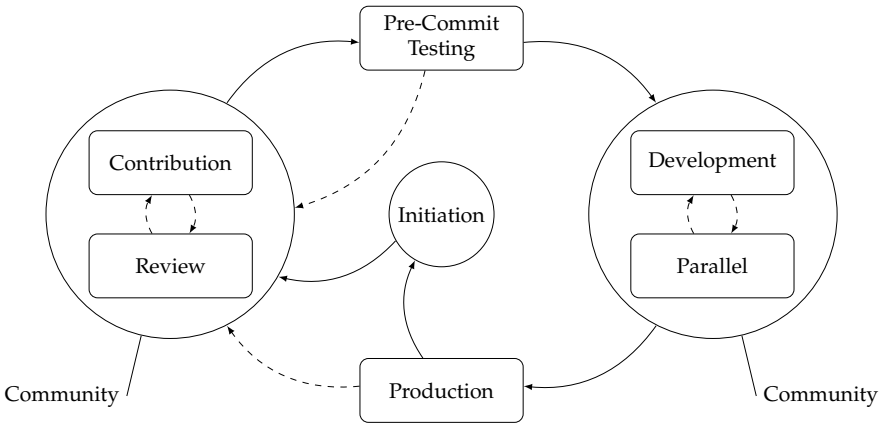


Figure 7.2: Life cycle model of Free and Open Source Software projects as proposed by Roets, Minnaar and Wright [RMW07].

Scrum terminology named sprints. Also Roets, Minnaar and Wright [RMW07] agree that no existing software engineering model resembles the development methods of FOSS projects accurately. They admittedly come up with a proposal of a FOSS development cycle which closely matches the given results. This proposal is based on the findings by Jørgensen [Jør01].

8 CONCLUSION

This thesis started with the problem and motivation of development processes in FOSS projects. It defined the main goal of the work, which was to analyze the development workflows of FOSS projects and to finally find matching patterns.

Next, the used theoretical background was explained and established. This included research methods such as the Grounded Theory or the Qualitative Content Analysis. The usage of such methods was vindicated by similar research papers and studies which also use same or similar research methods. Also traditional and agile software engineering methods were offered for later use. As traditional software engineering methods the Waterfall and Spiral model were explained. The instances of agile software engineering methods were the Extreme Programming and Scrum software engineering models.

The methodology of how projects were chosen was defined. Based on that, FOSS projects were analyzed and categorized to find matching specimens. The FOSS projects Debian, Drupal, Fedora, GNOME, KDE, MySQL/MariaDB, PHP, Plone, PostgreSQL and Python were chosen. Next, automated gathering of project data and the visualization thereof was discussed and explained. This determined the further analysis and the constitution of the work.

Data about the projects was collected and the chosen projects were analyzed in depth using the Grounded Theory research method. Six projects were examined thoroughly to identify a project analysis catalogue. Until this point the Grounded Theory approach was used. The catalogue was then applied to the other four projects with Qualitative Content Analysis, as proposed earlier.

Finally the outcome of the previous project analysis was compared. According to the project analysis catalogue differences and similarities between the projects were distinguished. This included all development related parts of the projects, such as the project origin, the community, communication and structure of the project, the release process with its schedules and cycles, as well as each project's development process with the general workflow, development lead and feature inclusion process.

The following discussion took the results of the previous analysis and compared it with similar researches or studies. High importance was given to a comparison with other research findings and whether they are similar or if the found results don't match with them.

As mentioned before the most important finding was that the project analysis catalogue was adequate for a development process anal-

ysis of the projects. Therefore the processes of the analyzed FOSS projects aren't that different after all. Of course, no project is equal to another one, but the general processes exist in one form or another in all other projects.

Similarities can be found already in the project's inception phases, which all started due to personal motivation. This generally concords with other findings. An unexpected result was that the Bazaar model could not be identified in any of the analyzed projects. Other researchers came to the same conclusion, however the Bazaar model is closely related to FOSS projects and therefore one would anticipate it in most of FOSS projects. On the contrary, all projects had a hierarchical structure, even if they were modular or had a flat structure. It has to be noted however, that only the core parts of the projects were analyzed and therefore, a Bazaar like model could still be identified around the projects. This was not analyzed in this thesis.

Despite the hierarchical structure of the projects, they are still very welcoming to new developers and often one can quickly progress and become a member in a leading group within the project. In this facet the Bazaar model is reaffirmed.

Furthermore there was a noticeable move towards fixed release cycles and more generally to an already pre-defined release schedule. This is very surprising as it is a widely spread opinion that FOSS projects do release when *it's ready*. This still can be seen in several projects, such as Debian or Drupal, however both projects have a strong opinion for fixed release cycles and it is only be a matter of time until they implement them.

Similar to commercial projects all analyzed projects had people or teams in place to ensure the compliance with the release schedule. This finding is also closely related to the previously noted hierarchical structure in the projects.

Another important finding is that the development processes of FOSS projects are similar between each other but cannot be resembled by traditional or agile software engineering methods. The development processes were all very agile and iterative. The outcome of this is obviously an evolutionary process in which the product gets better with each cycle. There are only a few cases known in which the project restarted from scratch. As such the processes and development methods found were quite different to the previously described software engineering methods. However several aspects of each proposed method can be found in the engineering cycles.

Finally the finding by Conway [Con68] can be corroborated and applied to the findings. It states that the organization of a software system is similar to the group which designed and implemented the system. According to the findings in this thesis this not only holds true for the final product but also for the development processes and workflows. The Python project stands as a primary example of a very

organized and structured group which also established a very formal and textured development process.

By no means this analysis can be considered as exhaustive, as the findings only hold for the chosen projects and their core parts. As such, it would be interesting to apply the project analysis catalogue to an extensive set of projects and analyze differences. A further analysis of such kind should of course establish different project selection criteria, such as less or more mature projects, but also smaller or larger projects than the analyzed ones.

Furthermore this analysis did not have a strong focus on the motivation of developers, the community or interaction thereof. These categories could provide material for an interesting study to see how they empower the different structures or workflows.

Also, as noted before, projects seem to evolve parallel to their processes over time. This raises a number of questions for future research such as if they evolve with a similar process or if there are similar processes or workflows to be found at similar times.

While the initial findings are promising, further research is necessary. This also includes the assets and drawbacks of fixed release cycles and why projects tend to adopt them.

Concluding this section, the work presented in this thesis provides a consistent framework for analysis and discussion of FOSS projects. Ideally it is able to describe processes adequately with a high focus on reusability and integrity. Free and Open Source Software projects are essential tools in the modern daily grind and we should be eager to see what else we can learn from Free and Open Source Software development processes and projects.

BIBLIOGRAPHY

[Arm06] D. J. Armstrong. "The Quarks of Object-Oriented Development". In: *Communications of the ACM* 49.2 (2006), pp. 123–128.

[Asp05] M. Aspeli. "Plone: A Model of a Mature Open Source Project". Master's Thesis. London School of Economics and Political Science, 2005.

[BA99] K. Beck and C. Andres. *Extreme Programming Explained: Embrace Change*. Vol. 1. Addison-Wesley, 1999, p. 224. ISBN: 0201616416.

[BCD03] D. Bainbridge, S. J. Cunningham and J. S. Downie. "How People Describe their Music Information Needs: A Grounded Theory Analysis of Music Queries". In: *Proceedings of the 4th International Conference on Music Information Retrieval*. 2003, pp. 221–222.

[Bec99] K. Beck. "Embracing Change with Extreme Programming". In: *IEEE Computer* 32.10 (1999), pp. 70–77.

[Bez99#1] N. Bezroukov. "A Second Look at the Cathedral and Bazaar". In: *First Monday* 4.12 (1999).

[Bez99#2] N. Bezroukov. "Open Source Software Development as a Special Type of Academic Research (Critique of Vulgar Raymondism)". In: *First Monday* 4.10 (1999).

[Boe88] B. W. Boehm. "A Spiral Model of Software Development and Enhancement". In: *IEEE Computer* 21.5 (1988), pp. 61–72.

[BT75] V. R. Basili and A. J. Turner. "Iterative Enhancement: A Practical Technique for Software Development". In: *IEEE Transactions on Software Engineering* SE-1.4 (1975), pp. 390–396.

[CH05] K. Crowston and J. Howison. "The Social Structure of Free and Open Source Software Development". In: *First Monday* 10.2 (2005).

[CM07] A. Capiluppi and M. Michlmayr. "From the Cathedral to the Bazaar: An Empirical Study of the Lifecycle of Volunteer Community Projects". In: *Open Source Development, Adoption and Innovation*. Vol. 234. Springer, 2007, pp. 31–44.

[Con68] M. E. Conway. "How Do Committees Invent?" In: *Datamation* 14.4 (1968), pp. 28–31.

[Cro+04] K. Crowston et al. "Towards A Portfolio of FLOSS Project Success Measures". In: *Collaboration, Conflict and Control: The 4th Workshop on Open Source Software Engineering, International Conference on Software Enginnering*. 2004.

[Cro+05] K. Crowston et al. "Effective Work Practices for FLOSS Development: A Model and Propositions". In: *Proceedings of the 38th Annual Hawaii International Conference on System Sciences*. IEEE Computer Society, 2005.

[DS90] P. DeGrace and L. H. Stahl. *Wicked Problems and Righteous Solutions: A Catalogue of Modern Software Engineering Paradigms*. Yourdon Press, 1990.

[DTB04] T. T. Dinh-Trong and J. M. Bieman. "Open Source Software Development: A Case Study of FreeBSD". In: *IEEE METRICS*. IEEE Computer Society, 2004, pp. 96–105.

[Ger03] D. M. German. "The GNOME Project: A Case Study of Open Source, Global Software Development". In: *Software Process: Improvement and Practice* 8.4 (2003), pp. 201–215.

[Gho05] R. Ghosh. "Understanding Free Software Developers: Findings from the FLOSS Study". In: *Perspectives on Free and Open Source Software* (2005), pp. 23–46.

[Gra09] J. Grazzini. "The Open Source Phenomenon". Working Paper. University of Turin, 2009.

[GS67] B. G. Glaser and A. L. Strauss. *The Discovery of Grounded Theory*. Vol. 20. Aldine de Gruyter, 1967, p. 271. ISBN: 0202302601.

[GT00] M. W. Godfrey and Q. Tu. "Evolution in Open Source Software: A Case Study". In: *The International Conference on Software Maintenance*. 2000.

[Haz+06] O. Hazzan et al. "Qualitative Research in Computer Science Education". In: *Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education* (2006), pp. 408–412.

[Hea04] H. Heath. "Developing a Grounded Theory Approach: A Comparison of Glaser and Strauss". In: *International Journal of Nursing Studies* 41.2 (2004), pp. 141–150.

[HNH03] G. Hertel, S. Niedner and S. Herrmann. "Motivation of Software Developers in Open Source Projects: An Internet Based Survey of Contributors to the Linux Kernel". In: *Research Policy* 32.7 (2003), pp. 1159–1177.

[Joh01] K. Johnson. "A Descriptive Process Model for Open-Source Software Development". Master's Thesis. University of Calgary, 2001.

[Jør01] N. Jørgensen. "Putting it All in the Trunk: Incremental Software Development in the FreeBSD Open Source Project". In: *Information Systems Journal* 11.4 (2001).

[Kim03] E. E. Kim. "An Introduction to Open Source Communities". Working Paper. Blue Oxen Associates, 2003.

[KM94] B. Kaplan and J. A. Maxwell. "Qualitative Research Methods for Evaluating Computer Information Systems". In: *Evaluating Health Care Information Systems Methods and Applications*. Vol. Part I. Health Informatics. Sage, 1994, pp. 45–68.

[Koc04] S. Koch. "Agile Principles and Open Source Software Development: A Theoretical and Empirical Discussion". In: *Extreme Programming and Agile Processes in Software Engineering*. Lecture Notes in Computer Science 3092. Springer, 2004, pp. 85–93.

[Kri02] S. Krishnamurthy. "Cave or Community? An Empirical Investigation of 100 Mature Open Source Projects". In: *First Monday* 7.6 (2002).

[KS02] S. Koch and G. Schneider. "Effort, Co-Operation and Co-Ordination in an Open Source Software Project: GNOME". In: *Information Systems Journal* 12.1 (2002), pp. 27–42.

[LH02] K. Lakhani and E. von Hippel. "How Open Source Software Works: "Free" User-to-User Assistance". In: *Research Policy* 32.6 (2002), pp. 923–943.

[LT00] J. Lerner and J. Tirole. *The Simple Economics of Open Source*. Tech. rep. 2000.

[LW03] K. Lakhani and R. G. Wolf. "Why Hackers Do What They Do: Understanding Motivation and Effort in Free/Open Source Software Projects". In: *Social Science Research Network* (2003), pp. 1–27.

[Mag10] H. Magnusson. "Community in Action". In: *php architect* (July 2010), pp. 35–40.

[May00] P. Mayring. "Qualitative Content Analysis". In: *Forum Qualitative Social Research* 1.2 (2000).

[May83] P. Mayring. *Qualitative Inhaltsanalyse. Grundlagen und Techniken*. Vol. 10. Beltz, 1983, p. 135. ISBN: 978-3-407-25501-3.

[MFH02] A. Mockus, R. T. Fielding and J. D. Herbsleb. "Two Case Studies of Open Source Software Development: Apache and Mozilla". In: *ACM Transactions on Software Engineering and Methodology* 11.3 (2002), pp. 309–346.

[MSABA10] O. Meerbaum-Salant, M. Armoni and M. M. Ben-Ari. "Learning Computer Science Concepts with Scratch". In: *Proceedings of the Sixth International Workshop on Computing Education Research* (2010), pp. 69–76.

[Oga+07] M. Ogawa et al. "Visualizing Social Interaction in Open Source Software Projects". In: *APVIS*. IEEE Computer Society, 2007, pp. 25–32.

[Pan96] N. R. Pandit. "The Creation of Theory: A Recent Application of the Grounded Theory Method". In: *The Qualitative Report* 2.4 (1996).

[PPV00] D. E. Perry, A. A. Porter and L. G. Votta. "Empirical Studies of Software Engineering: A Roadmap". In: *Proceedings of the Conference on the Future of Software Engineering*. ACM. 2000, pp. 345–355.

[Ray98] E. Raymond. "The Cathedral and the Bazaar". In: *First Monday* 3.3 (1998).

[RMW07] R. Roets, M. Minnaar and K. Wright. "Open Source: Towards Successful Systems Development Projects in Developing Countries". In: *Computers in Developing Countries* (2007).

[Roy70] W. W. Royce. "Managing the Development of Large Software Systems". In: *Proceedings of IEEE WESCON* (1970), pp. 1–9.

[SC90] A. L. Strauss and J. M. Corbin. *Basics of Qualitative Research: Grounded Theory Procedures and Techniques*. Vol. 2. Sage, 1990, p. 270. ISBN: 0803932502.

[Sca06] W. Scacchi. "Understanding Open-Source Software Evolution". In: *Software Evolution and Feedback: Theory and Practice*. 2006, pp. 181–205.

[Sch95] K. Schwaber. "Scrum Development Process". In: *OOPSLA Business Object Design and Implementation* (1995), pp. 10–19.

[SLS01] S. Sarker, F. Lau and S. Sahay. "Using an Adapted Grounded Theory Approach for Inductive Theory Building about Virtual Team Development". In: *Database for Advances in Information Systems* 32.1 (2001), pp. 38–56.

[SS03] C. M. Schweik and A. Semenov. "The Institutional Design of Open Source Programming: Implications for Addressing Complex Public Policy and Management Problems". In: *First Monday* 8.1 (2003).

[SS04] D. Spinellis and C. Szyperski. "How is Open Source Affecting Software Development?" In: *IEEE Software* 21.1 (2004), pp. 28–33.

[SSRD08] B. M. Sadowski, G. Sadowski-Rasters and G. Duysters. "Transition of Governance in a Mature Open Software Source Community: Evidence from the Debian Case". In: *Information Economics and Policy* 20.4 (2008), pp. 323–332.

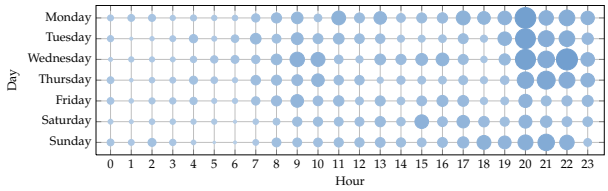
[Sut95] J. Sutherland. "Business Object Design and Implementation Workshop". In: *ACM SIGPLAN OOPS Messenger* 6.4 (1995), pp. 170–175.

[WA03] J. Warsta and P. Abrahamsson. "Is Open Source Software Development Essentially an Agile Method?" In: *Proceedings of the 3rd Workshop on Open Source Software Engineering 25th International Conference on Software Engineering*. 2003, pp. 143–147.

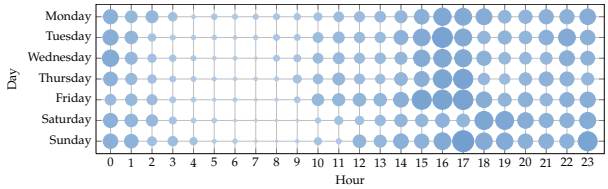
APPENDIX

A

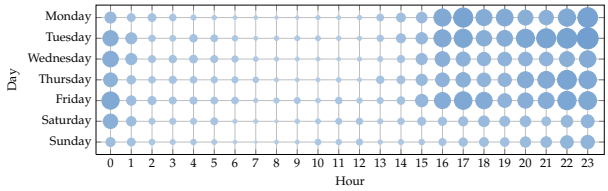
COMPARISON OF PROJECT GRAPHS



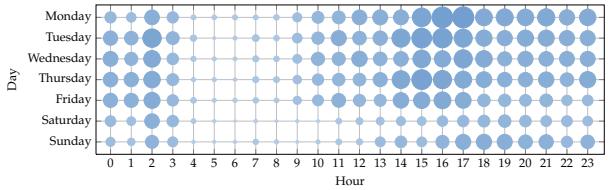
(a) Drupal, figure 5.4 on page 25.



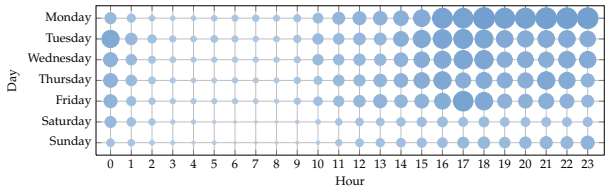
(b) Plone, figure 5.10 on page 30.



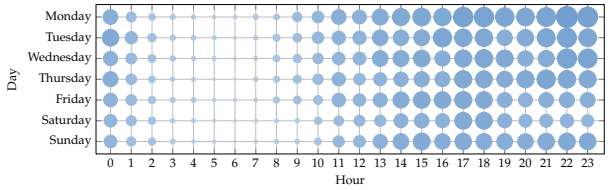
(c) Python, figure 5.17 on page 36.



(d) PHP, figure 5.25 on page 43.

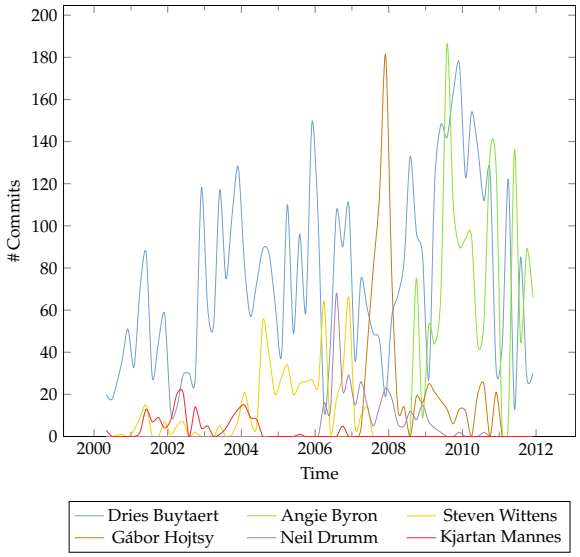


(e) GNOME, figure 5.36 on page 52.

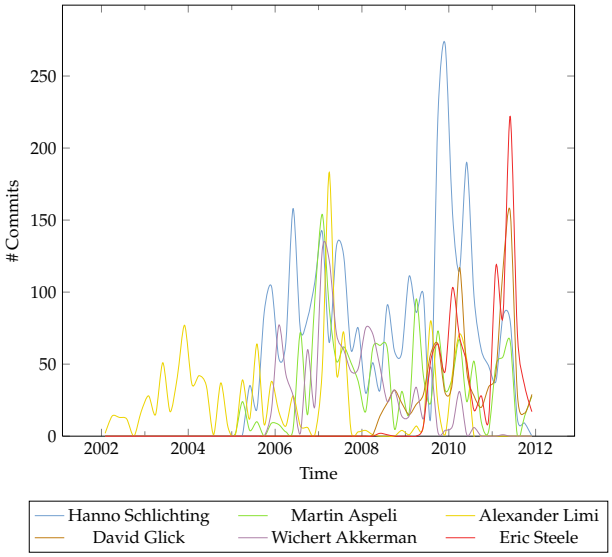


(f) KDE, figure 5.43 on page 60.

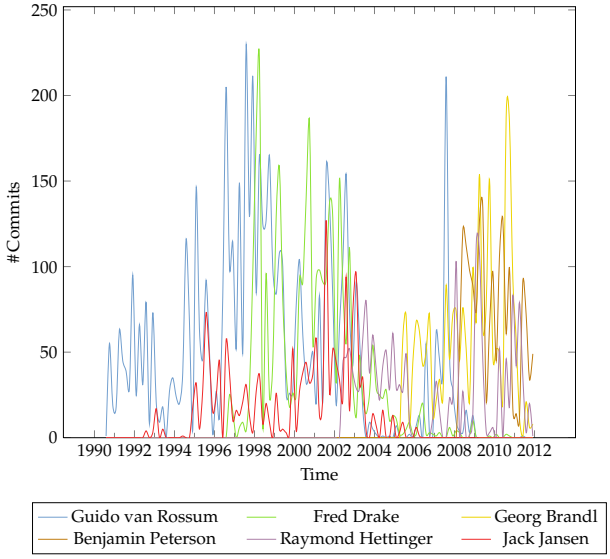
Figure A.1: Overview of time based view graphs.



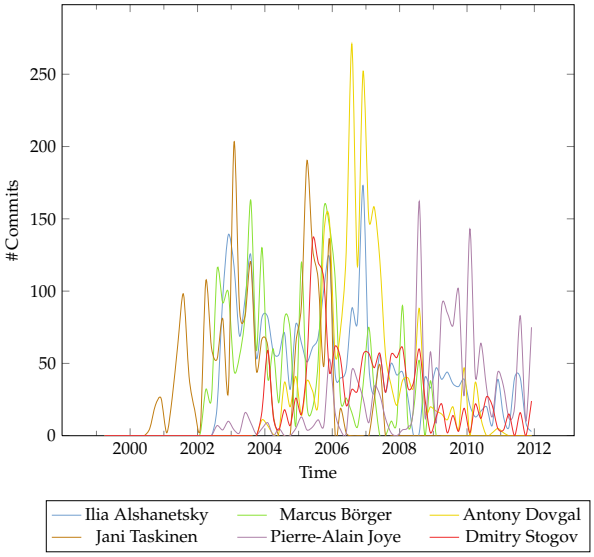
(a) Drupal, figure 5.1 on page 22.



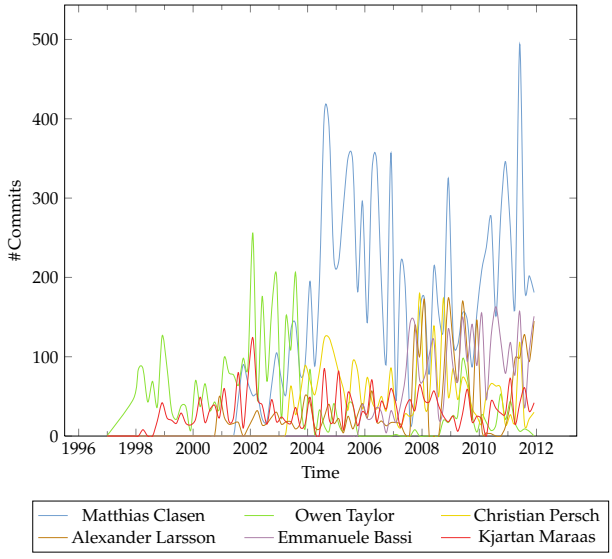
(b) Plone, figure 5.8 on page 29.



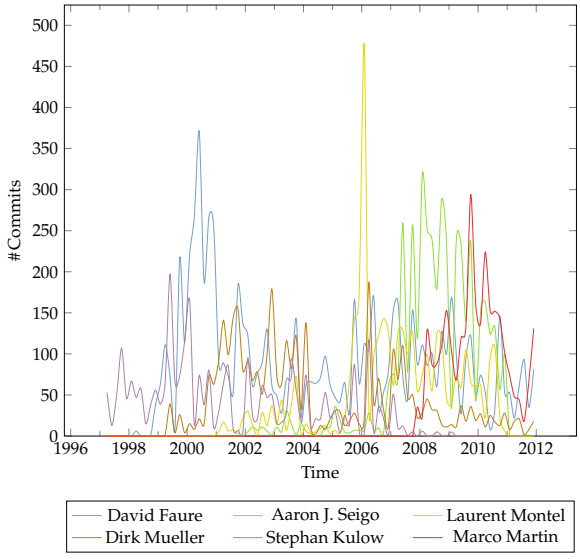
(c) Python, figure 5.15 on page 35.



(d) PHP, figure 5.23 on page 42.

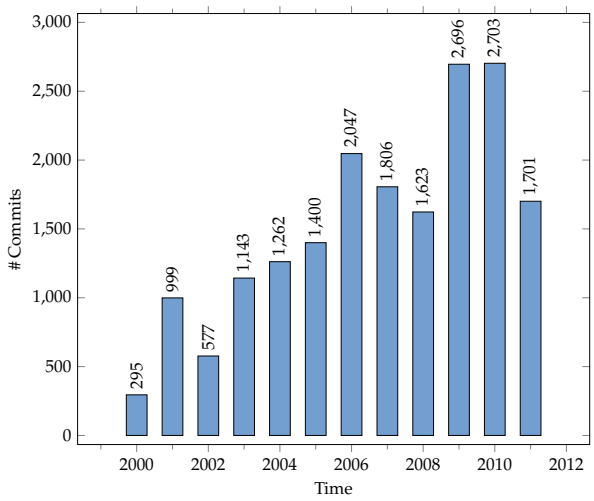


(e) GNOME, figure 5.31 on page 48.

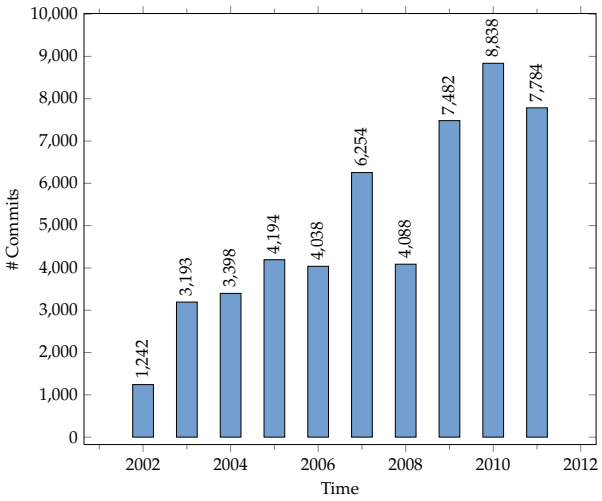


(f) KDE, figure 5.38 on page 57.

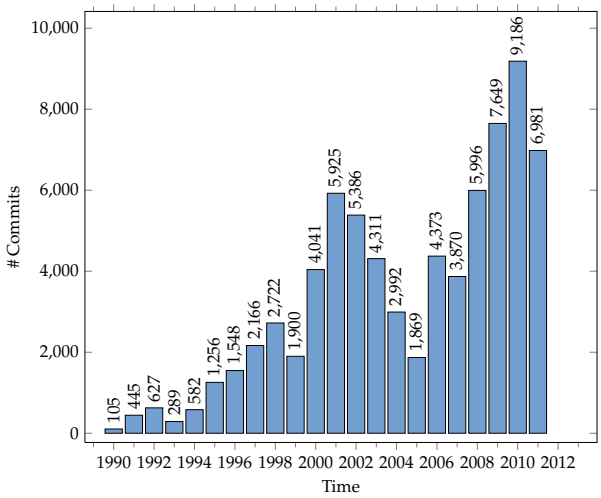
Figure A.2: Overview of commits by author graphs.



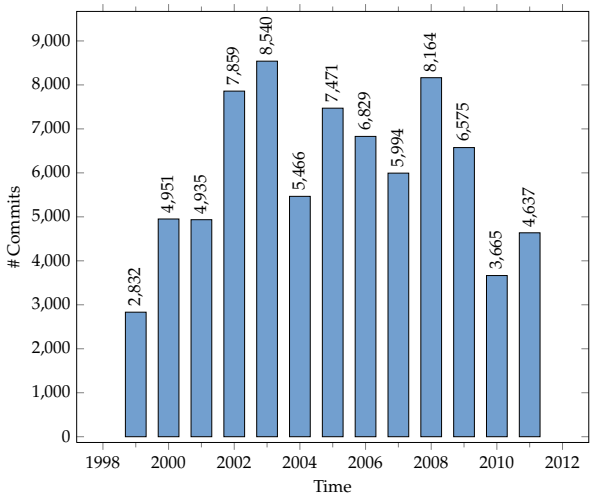
(a) Drupal, figure 5.2 on page 24.



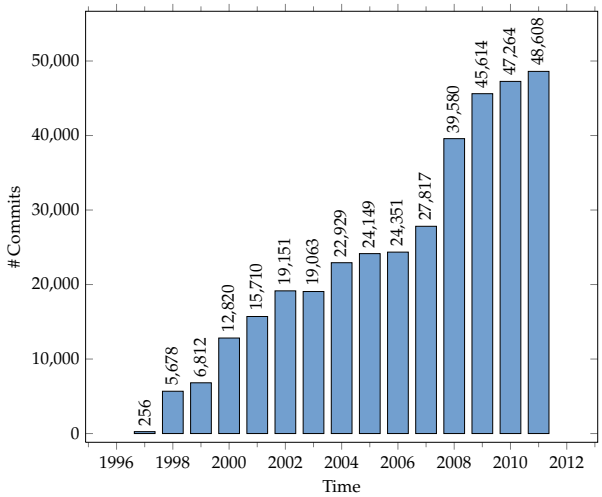
(b) Plone, figure 5.9 on page 30.



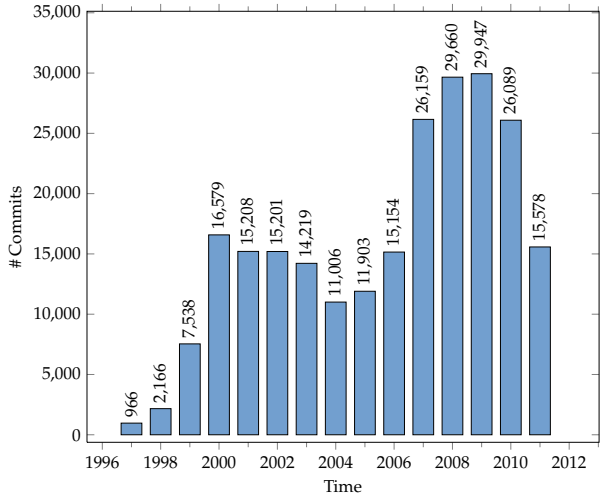
(c) Python, figure 5.16 on page 36.



(d) PHP, figure 5.24 on page 43.

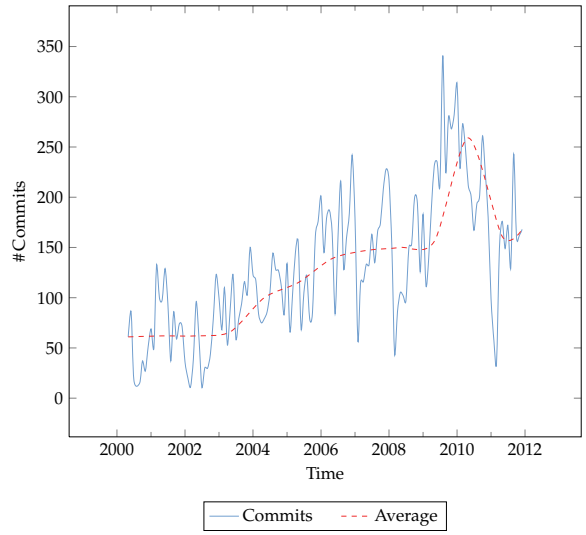


(e) GNOME, figure 5.32 on page 50.

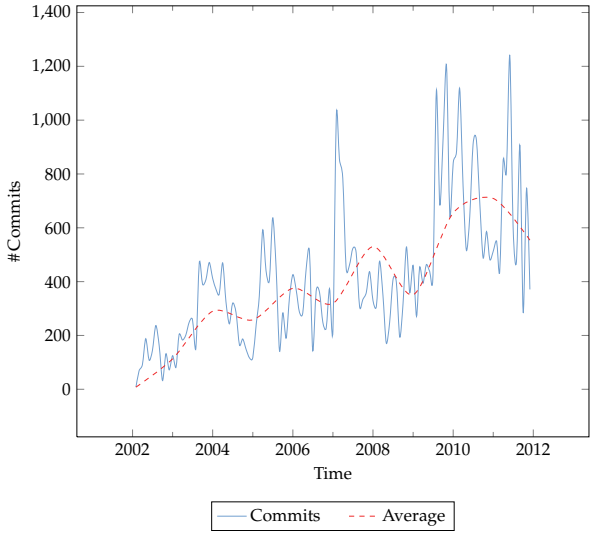


(f) KDE, figure 5.39 on page 58.

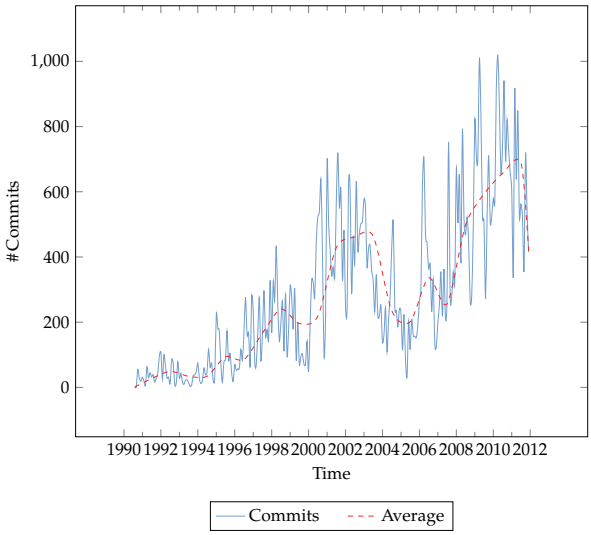
Figure A.3: Overview of commits by year graphs.



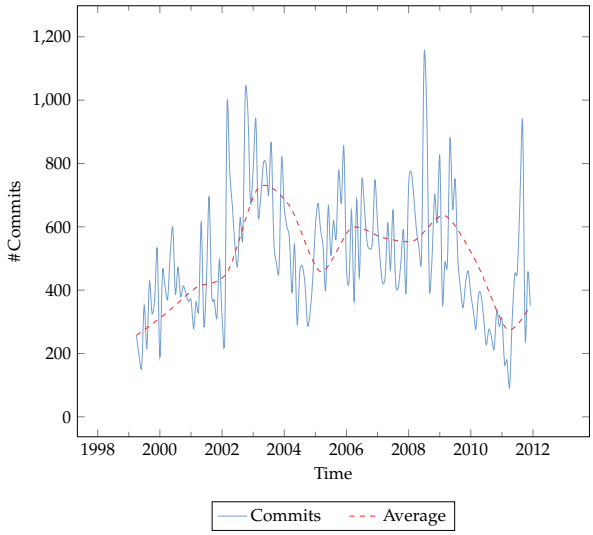
(a) Drupal, figure 5.6 on page 26.



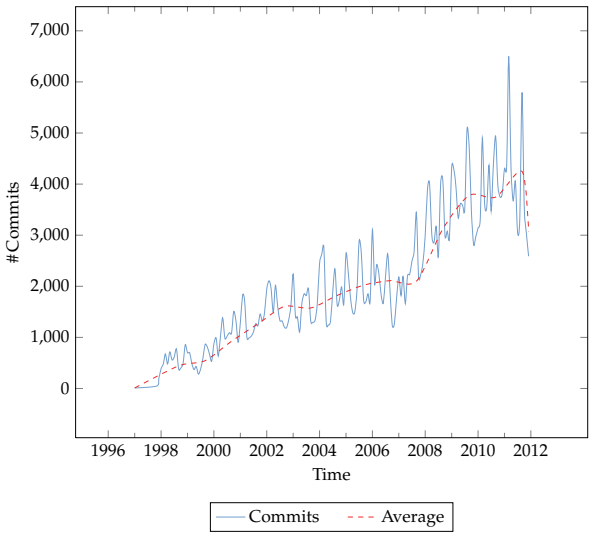
(b) Plone, figure 5.12 on page 32.



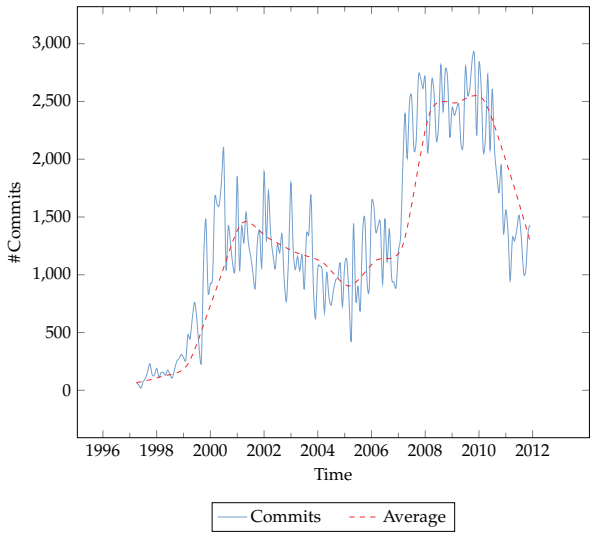
(c) Python, figure 5.18 on page 37.



(d) PHP, figure 5.28 on page 45.

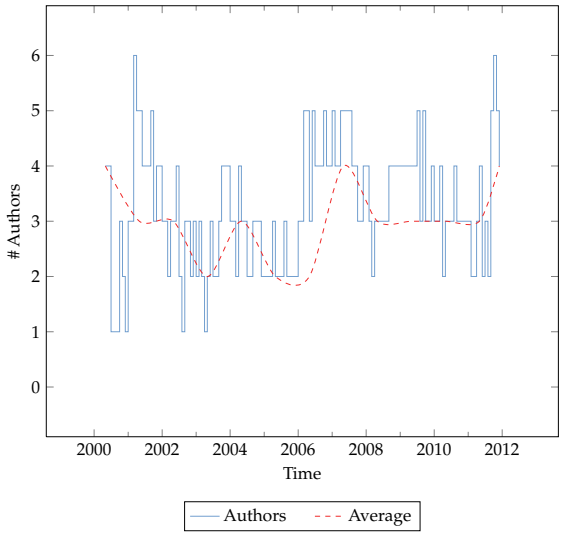


(e) GNOME, figure 5.35 on page 52.

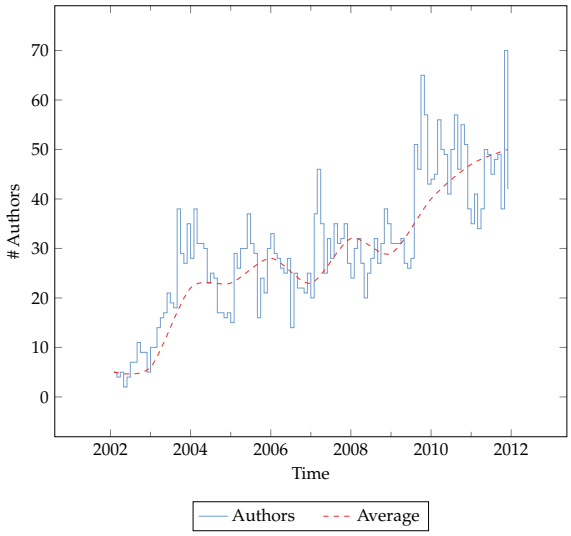


(f) KDE, figure 5.41 on page 59.

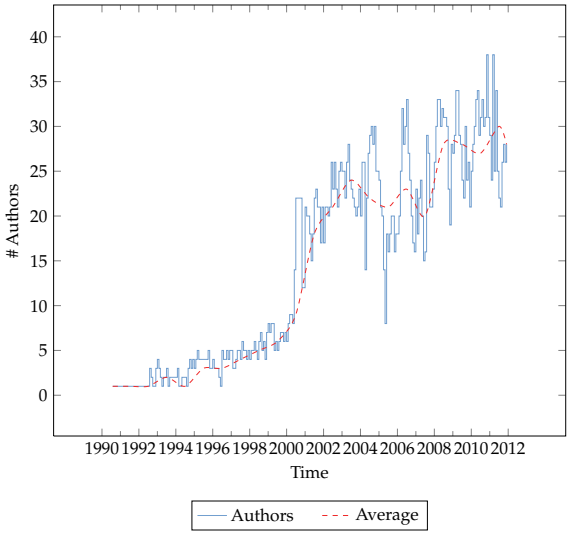
Figure A.4: Overview of commits by month graphs.



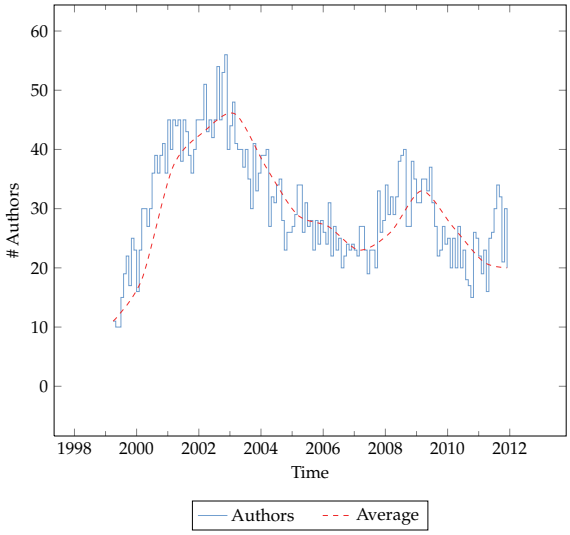
(a) Drupal, figure 5.7 on page 27.



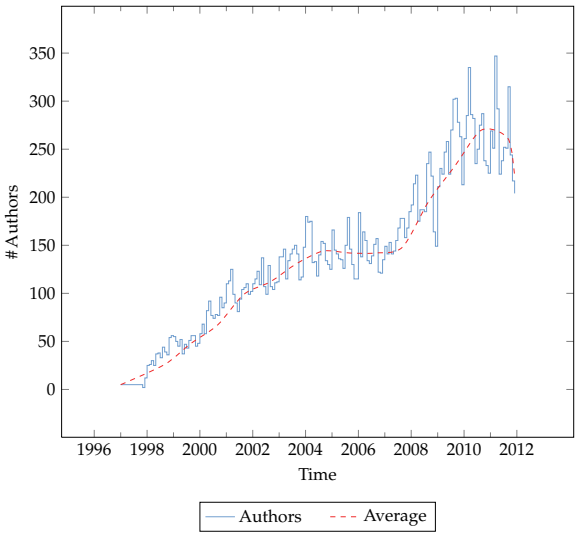
(b) Plone, figure 5.14 on page 33.



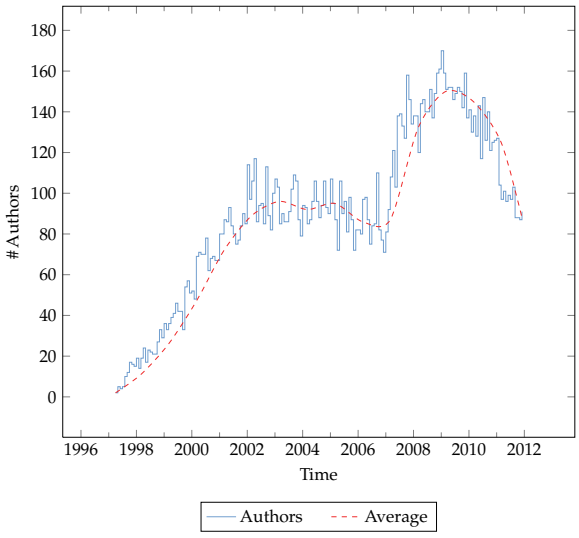
(c) Python, figure 5.21 on page 39.



(d) PHP, figure 5.30 on page 47.



(e) GNOME, figure 5.37 on page 53.



(f) KDE, figure 5.42 on page 60.

Figure A.5: Overview of authors by month graphs.

B

PROJECT RESOURCES

All stated websites were last checked on December 31st, 2011. Whenever possible, the publication date was added to the citation.

B.1 DRUPAL

[Bui] BuiltWith. *Drupal Usage Trends*. URL: http://trends.builtwith.com/cms/Drupal.

[Buy11] D. Buytaert. *Drupal Contributor Statistics*. June 2011. URL: http://buytaert.net/drupal-contributor-statistics-2011.

[W3T#1] W3Techs. *Usage Statistics and Market Share of Content Management Systems for Websites*. URL: http://w3techs.com/technologies/overview/content_management/all.

[Wal11] J. Walling. *Drupal Events*. Nov. 2011. URL: http://www.listology.com/jwalling/list/drupal-events.

[Dru#1] Drupal Project. *Core Developers*. URL: http://drupal.org/node/21778.

[Dru#2] Drupal Project. *Drupal Core Initiatives*. URL: http://drupal.org/community-initiatives/drupal-core.

[Dru#3] Drupal Project. *Drupal Core's Release Cycle*. URL: http://drupal.org/node/935558.

[Dru#4] Drupal Project. *History*. URL: http://drupal.org/about/history.

[Dru#5] Drupal Project. *The Drupal Overview*. URL: http://drupal.org/node/265726.

[Dru#6] Drupal Project. *Upgrading from Previous Versions*. URL: http://drupal.org/upgrade.

B.2 PLONE

[Asp05] M. Aspeli. "Plone: A Model of a Mature Open Source Project". Master's Thesis. London School of Economics and Political Science, 2005.

[Bla11] Black Duck Software Inc. *Ohloh Plone Factoids*. 2011. URL: http://www.ohloh.net/p/plone/factoids.

[Plo#1] Plone Project. *Community Processes*. URL: http://dev.plone.org/wiki/CommunityProcesses.

[Plo#2] Plone Project. *How You Can Contribute to Plone*. URL: http://dev.plone.org/wiki/ContributePlone.

[Plo#3] Plone Project. *PLIP Lifecycle*. URL: http://dev.plone.org/wiki/PLIPLifecycle.

[Plo#4] Plone Project. *PLIP Process*. URL: http://dev.plone.org/wiki/PlipProcess.

[Plo#5] Plone Project. *Plone Conferences*. URL: http://plone.org/events/conferences.

[Plo#6] Plone Project. *Plone FAQ*. URL: http://plone.org/documentation/faq.

[Plo#7] Plone Project. *Plone Founders*. URL: http://plone.org/team.

[Plo#8] Plone Project. *Plone Framework Team*. URL: http://dev.plone.org/wiki/FrameworkTeam.

[Plo#9] Plone Project. *Plone Release Managers*. URL: http://plone.org/team/ReleaseManagers.

[Plo#10] Plone Project. *Plone Release Process*. URL: http://dev.plone.org/wiki/ReleaseProcess.

[Plo#11] Plone Project. *Plone Sprints*. URL: http://plone.org/events/sprints.

[Plo#12] Plone Project. *Releases*. URL: http://plone.org/products/plone/releases.

[Plo#13] Plone Project. *What is Plone?* URL: http://plone.org/about/.

[Plo04] Plone Project. *Plone Gets Paid Release Manager*. Oct. 2004. URL: http://plone.org/news/plone-gets-paid-release-management.

[Plo11] Plone Project. *Plone Alters PLIP Process to Implement a 6 Month Fixed Release Cycle*. Mar. 2011. URL: http://plone.org/news/newsixmonthreleasecycle.

B.3 PYTHON

[Ven03] B. Venners. *The Making of Python - A Conversation with Guido van Rossum, Part I*. Jan. 2003. URL: http://www.artima.com/intv/pythonP.html.

[War02] B. Warsaw. *Voting Guidelines – PEP 10*. Mar. 2002. URL: http://www.python.org/dev/peps/pep-0010/.

[WHG00] B. Warsaw, J. Hylton and D. Goodger. *PEP Purpose and Guidelines – PEP 1*. June 2000. URL: http://www.python.org/dev/peps/pep-0001/.

[WR01] B. Warsaw and G. van Rossum. *Doing Python Releases – PEP 101*. Aug. 2001. URL: http://www.python.org/dev/peps/pep-0101/.

[Pyt#1] Python Software Foundation. *Conferences and Workshops*. URL: http://www.python.org/community/workshops/.

[Pyt#2] Python Software Foundation. *Development Cycle*. URL: http://docs.python.org/devguide/devcycle.html.

[Pyt#3] Python Software Foundation. *Following Python's Development*. URL: http://docs.python.org/devguide/communication.html.

[Pyt#4] Python Software Foundation. *History and License*. URL: http://docs.python.org/license.html.

[Pyt#5] Python Software Foundation. *How to Become a Core Developer*. URL: http://docs.python.org/devguide/coredev.html.

[Pyt#6] Python Software Foundation. *What is Python? Executive Summary*. URL: http://www.python.org/doc/essays/blurb/.

B.4 PHP

[Mag10] H. Magnusson. "Community in Action". In: *php architect* (July 2010), pp. 35–40.

[W3T#2] W3Techs. *Usage Statistics and Market Share of Server-Side Programming Languages for Websites*. URL: http://w3techs.com/technologies/overview/programming_language/all.

[PHP#1] PHP Group. *Credits*. URL: http://www.php.net/credits.php.

[PHP#2] PHP Group. *History of PHP*. URL: http://www.php.net/manual/en/history.php.php.

[PHP#3] PHP Group. *PHP Conferences Around the World*. URL: http://www.php.net/conferences/index.php.

[PHP#4] PHP Group. *PHP Manual*. URL: http://www.php.net/manual/en/index.php.

[PHP#5] PHP Group. *Request for Comments*. URL: https://wiki.php.net/rfc.

[PHP#6] PHP Group. *RFC Release Process*. URL: https://wiki.php.net/rfc/releaseprocess.

[PHP#7] PHP Group. *RFC Voting on PHP features*. URL: https://wiki.php.net/rfc/voting.

[PHP#8] PHP Group. *RFC Who can Vote?* URL: https://wiki.php.net/rfc/voting_who.

[PHP#9] PHP Group. *What can PHP do?* URL: http://www.php.net/manual/en/intro-whatcando.php.

[PHP07] PHP Group. *PHP Usage Stats*. July 2007. URL: http://www.php.net/usage.php.

B.5 GNOME

[Ger03] D. M. German. "The GNOME Project: A Case Study of Open Source, Global Software Development". In: *Software Process: Improvement and Practice* 8.4 (2003), pp. 201–215.

[GNO#1] GNOME Project. *About GUADEC 2012*. URL: http://guadec.org/?q=node/3.

[GNO#2] GNOME Project. *About Us*. URL: http://www.gnome.org/about/.

[GNO#3] GNOME Project. *Design Team*. URL: https://live.gnome.org/Design.

[GNO#4] GNOME Project. *GNOME 3.3.x Development Series*. URL: https://live.gnome.org/ThreePointThree.

[GNO#5] GNOME Project. *GNOME Human Interface Guidelines*. URL: http://developer.gnome.org/hig-book/.

[GNO#6] GNOME Project. *GNOME Release Schedule*. URL: https://live.gnome.org/Schedule.

[GNO#7] GNOME Project. *Guide for New Release Team Members*. URL: https://live.gnome.org/ReleasePlanning/NewReleaseTeamMembers.

[GNO#8] GNOME Project. *Proposed Features for GNOME 3.4*. URL: https://live.gnome.org/ThreePointThree/Features.

[GNO#9] GNOME Project. *Road Map*. URL: https://live.gnome.org/RoadMap.

[GNO#10] GNOME Project. *Teams*. URL: http://www.gnome.org/teams/.

[GNO11] GNOME Project. *GNOME 3.0 released*. Apr. 2011. URL: http://www.gnome.org/press/2011/04/gnome-3-0-released-better-for-users-developers-3/.

[GNO97] GNOME Project. *The GNOME Desktop Project*. Aug. 1997. URL: http://mail.gnome.org/archives/gtk-list/1997-August/msg00123.html.

B.6 KDE

[KDE#1] KDE Project. *About KDE*. URL: http://kde.org/community/whatiskde/.

[KDE#2] KDE Project. *Contribute*. URL: http://techbase.kde.org/Contribute.

[KDE#3] KDE Project. *Development Model*. URL: http://kde.org/community/whatiskde/devmodel.php.

[KDE#4] KDE Project. *General FAQ*. URL: http://techbase.kde.org/Development/FAQs/General_FAQ.

[KDE#5] KDE Project. *Get a Contributor Account*. URL: http://techbase.kde.org/Contribute/Get_a_Contributor_Account.

[KDE#6] KDE Project. *KDE 4.8 Release Schedule*. URL: http://techbase.kde.org/Schedules/KDE4/4.8_Release_Schedule.

[KDE#7] KDE Project. *KDE History*. URL: http://kde.org/community/history/.

[KDE#8] KDE Project. *KDE Software Compilation*. URL: http://kde.org/community/whatiskde/softwarecompilation.php.

[KDE#9] KDE Project. *Press Page*. URL: http://kde.org/presspage/.

[KDE#10] KDE Project. *Project Management*. URL: http://kde.org/community/whatiskde/management.php.

[KDE#11] KDE Project. *Release Team*. URL: http://techbase.kde.org/Projects/Release_Team.

[KDE#12] KDE Project. *Schedules*. URL: http://techbase.kde.org/Schedules.

[KDE96] KDE Project. *KDE Project Announcement*. Oct. 1996. URL: http://kde.org/documentation/posting.txt.

B.7 POSTGRESQL

[Pos#1] PostgreSQL Global Development Group. *About*. URL: http://www.postgresql.org/about/.

[Pos#2] PostgreSQL Global Development Group. *Awards*. URL: http://www.postgresql.org/about/awards/.

[Pos#3] PostgreSQL Global Development Group. *Commit Fest*. URL: http://wiki.postgresql.org/wiki/CommitFest.

[Pos#4] PostgreSQL Global Development Group. *Contributor Profiles*. URL: http://www.postgresql.org/community/contributors/.

[Pos#5] PostgreSQL Global Development Group. *Developer FAQ*. URL: http://wiki.postgresql.org/wiki/Developer_FAQ.

[Pos#6] PostgreSQL Global Development Group. *Downloads*. URL: http://www.postgresql.org/download/.

[Pos#7] PostgreSQL Global Development Group. *Events*. URL: http://www.postgresql.org/about/eventarchive/.

[Pos#8] PostgreSQL Global Development Group. *Frequently Asked Press Questions*. URL: http://www.postgresql.org/about/press/faq/.

[Pos#9] PostgreSQL Global Development Group. *History*. URL: http://www.postgresql.org/about/history/.

[Pos#10] PostgreSQL Global Development Group. *License*. URL: http://www.postgresql.org/about/licence/.

[Pos#11] PostgreSQL Global Development Group. *PostgreSQL 9.1 Press Kit*. URL: http://www.postgresql.org/about/press/presskit91/.

[Pos#12] PostgreSQL Global Development Group. *PostgreSQL 9.2 Development Plan*. URL: http://wiki.postgresql.org/wiki/PostgreSQL_9.2_Development_Plan.

[Pos#13] PostgreSQL Global Development Group. *Running a Commit Fest*. URL: http://wiki.postgresql.org/wiki/Running_a_CommitFest.

[Pos#14] PostgreSQL Global Development Group. *Versioning Policy*. URL: http://www.postgresql.org/support/versioning/.

B.8 MYSQL/MARIADB

[Mon#1] Monty Program Ab. *About MariaDB*. URL: http://kb.askmonty.org/en/about-mariadb.

[Mon#2] Monty Program Ab. *Contributing Code*. URL: http://kb.askmonty.org/en/contributing-code.

[Mon#3] Monty Program Ab. *Contributing to the MariaDB Project*. URL: http://kb.askmonty.org/en/community-contributing-to-the-mariadb-project.

[Mon#4] Monty Program Ab. *MariaDB Roadmap*. URL: http://kb.askmonty.org/en/mariadb-roadmap.

[Mon#5] Monty Program Ab. *Plans for 5.6*. URL: http://kb.askmonty.org/en/plans-for-56.

[Mon#6] Monty Program Ab. *Release Coordinator*. URL: http://kb.askmonty.org/en/release-coordinator.

[Mon#7] Monty Program Ab. *Release Criteria*. URL: http://kb.askmonty.org/en/release-criteria.

[Mon#8] Monty Program Ab. *What Are the Criteria for Becoming a Maria-Captain?* URL: http://kb.askmonty.org/en/what-are-the-criteria-for-becoming-a-maria-captain.

[Mon#9] Monty Program Ab. *What is MariaDB 5.1*. URL: http://kb.askmonty.org/en/what-is-mariadb-51.

[Mon#10] Monty Program Ab. *What License Does MariaDB Use?* URL: http://kb.askmonty.org/en/what-license-does-mariadb-use.

[Mon#11] Monty Program Ab. *Where Are Other Users and Developers of MariaDB?* URL: http://kb.askmonty.org/en/where-are-other-users-and-developers-of-mariadb.

[Mon#12] Monty Program Ab. *Who is Behind MariaDB?* URL: http://kb.askmonty.org/en/who-is-behind-mariadb.

[Ora#1] Oracle Corporation. *Choosing Which Version of MySQL to Install*. URL: http://dev.mysql.com/doc/refman/5.1/en/choosing-version.html.

[Ora#2] Oracle Corporation. *History of MySQL*. URL: http://dev.mysql.com/doc/refman/5.6/en/history.html.

[Ora#3] Oracle Corporation. *MySQL Users Conference & Expo*. URL: http://www.mysql.com/news-and-events/users-conference/2010/.

[Ora10] Oracle Corporation. *Oracle Completes Acquisition of Sun*. Jan. 2010. URL: http://www.oracle.com/us/corporate/press/044428.

[Sun08] Sun Microsystems Inc. *Sun to Acquire MySQL*. Jan. 2008. URL: http://www.mysql.com/news-and-events/sun-to-acquire-mysql.html.

B.9 FEDORA

[Fed#1] Fedora Project. *Board*. URL: http://fedoraproject.org/wiki/Board.

[Fed#2] Fedora Project. *Communicating and Getting Help*. URL: http://fedoraproject.org/wiki/Communicate.

[Fed#3] Fedora Project. *Features Policy Guidelines*. URL: http://fedoraproject.org/wiki/Features/Policy.

[Fed#4] Fedora Project. *Fedora Engineering Steering Committee*. URL: http://fedoraproject.org/wiki/Fedora_Engineering_Steering_Committee.

[Fed#5] Fedora Project. *Fedora Release Life Cycle*. URL: http://fedoraproject.org/wiki/LifeCycle.

[Fed#6] Fedora Project. *Fedora Weekly Newsletter*. URL: http://fedoraproject.org/wiki/FWN.

[Fed#7] Fedora Project. *FUDCon*. URL: http://fedoraproject.org/wiki/FUDCon.

[Fed#8] Fedora Project. *Historical Fedora Release Schedules*. URL: http://fedoraproject.org/wiki/Releases/HistoricalSchedules.

[Fed#9] Fedora Project. *Join*. URL: https://fedoraproject.org/wiki/Join.

[Fed#10] Fedora Project. *Licensing Guidelines*. URL: https://fedoraproject.org/wiki/Licensing.

[Fed#11] Fedora Project. *Release Engineering*. URL: http://fedoraproject.org/wiki/ReleaseEngineering/Overview.

[Fed#12] Fedora Project. *SIGs*. URL: http://fedoraproject.org/wiki/SIGs.

[Fed#13] Fedora Project. *Statistics*. URL: http://fedoraproject.org/wiki/Statistics#Contributors.

[Fed#14] Fedora Project. *Warren Togami Biography*. URL: http://fedoraproject.org/wiki/User:Wtogami.

[Fed#15] Fedora Project. *What is Fedora and what Makes it Different*. URL: https://fedoraproject.org/about-fedora.

B.10 DEBIAN

[McG11] N. McGovern. *Bits from the Release Team*. June 2011. URL: http://lists.debian.org/debian-devel-announce/2011/06/msg00003.html.

[Per11] C. Perrier. *Developers per Country*. June 2011. URL: http://www.perrier.eu.org/weblog/2011/06/12#devel-countries-201106-3.

[SSRD08] B. M. Sadowski, G. Sadowski-Rasters and G. Duysters. "Transition of Governance in a Mature Open Software Source Community: Evidence from the Debian Case". In: *Information Economics and Policy* 20.4 (2008), pp. 323–332.

[Deb#1] Debian Project. *A Brief History of Debian*. URL: http://www.debian.org/doc/manuals/project-history/.

[Deb#2] Debian Project. *About Debian*. URL: http://www.debian.org/intro/about.

[Deb#3] Debian Project. *DebConf*. URL: http://debconf.org/.

[Deb#4] Debian Project. *Debian Developer*. URL: http://wiki.debian.org/DebianDeveloper.

[Deb#5] Debian Project. *Debian Maintainer*. URL: http://wiki.debian.org/DebianMaintainer.

[Deb#6] Debian Project. *Debian New Members Corner*. URL: http://www.debian.org/devel/join/newmaint.

[Deb#7] Debian Project. *Debian Release Management*. URL: http://release.debian.org/.

[Deb#8] Debian Project. *Debian Releases*. URL: http://www.debian.org/releases/.

[Deb#9] Debian Project. *Debian Voting Information*. URL: http://www.debian.org/vote/.

[Deb#10] Debian Project. *Debian's Organizational Structure*. URL: http://www.debian.org/intro/organization.

[Deb#11] Debian Project. *License information*. URL: http://www.debian.org/legal/licenses/index.en.html.

[Deb#12] Debian Project. *Mailing Lists*. URL: http://www.debian.org/MailingLists/.

[Deb#13] Debian Project. *Ports*. URL: http://www.debian.org/ports/.

[Deb#14] Debian Project. *The Debian GNU/Linux FAQ*. URL: http://www.debian.org/doc/manuals/debian-faq/.

[Deb09] Debian Project. *Debian GNU/Linux 6.0 "Squeeze" Release Goals*. July 2009. URL: http://www.debian.org/News/2009/20090730.en.html.