Abstract

While many successful and well-known open source projects produce output of high quality, a general assessment of this development paradigm is still missing. In this paper, an online community of both small and large, successful and failed projects and their programmers is analysed mainly using the version-control data of each project, also according to their productivity and estimation of expended effort. As the results show, there are indeed significant differences between this cooperative development model and the commercial organization of work in the areas explored. Both open source software projects in their size and their programmers’ effort differ significantly, and the evolution of projects’ size over time seems in part to contradict the laws of software evolution proposed for commercial systems. Both the inequality of effort distribution between programmers and an increasing number of developers in a project do not lead to a decrease in productivity, opposing Brooks’s Law. Effort estimation based on the COCOMO model for commercial organizations shows a large amount of effort expended for the projects, while a more general Norden–Rayleigh modeling shows a distinctly smaller expenditure. This proposes that either a highly efficient development is achieved by this self-organizing cooperative and highly decentralized form of work, or that the participation of users besides programming tasks is enormous and constitutes an economic factor of large proportions.

Keywords: open source, software development, software metrics, productivity, effort estimation

INTRODUCTION

Over the past years, free and open source software has gathered increasing interest, both from the business and academic world. As some projects in different application domains like Linux together with the suite of GNU utilities, GNOME, KDE, Apache, sendmail, bind, and several programming languages have achieved huge success in their respective markets, new business models have been developed and tested by businesses both small and large like Netscape or IBM. Academic interest into this new form of collaborative software development has arisen from very different backgrounds including software engineering, sociology, management or psychology, and has gained increasing prominence, as can be deduced from the number of international journals like Management Science, Information Systems Journal or Research Policy, and conferences like ICSE dedicating special issues, workshops and tracks to this new field of research.

While many of the successful projects mentioned above are well-known and produce output of high quality, a general assessment of the open source software development paradigm remains unavailable. Currently, any discussion of this new model is mostly based on a small number of glamorous and successful projects — but these might constitute exceptions. Therefore, an analysis needs to be based on quantitative information on the enactment of this new model, especially in a variety of project forms and sizes, including as diverse a population as possible. The main ideas of this development model are described in the seminal work of Raymond (1999), The Cathedral and the Bazaar, in which he contrasts the traditional type of software development of a few people planning a cathedral in splendid isolation with the new collaborative bazaar form of open source software development. In this, a large number of developer-turned users come together without monetary compensation to cooperate under a model of rigorous peer-review and take advantage of parallel debugging that leads to innovation and rapid advancement in developing and evolving software products, thus forming an example for ‘egoless programming’ as proposed by Weinberg in 1971 (Weinberg 1998). In order to allow for this to happen and to minimize duplicated work, the source code of the software needs to be accessible, and new versions need to be released often. To this end, software licences that grant the necessary rights to the users, like free redistribution, inclusion of the source code, the possibility for modifications and derived works and...
of effort between participants, the distribution of project sizes and the effects of both team size and distribution on productivity, most notably exploring the presence or absence of the well-known Brooks’s Law (Brooks 1995). In addition, the view that several commercial estimation models like COCOMO (Boehm 1981) offer on the effort expended by the participants of open source development will be explored, contrasting these findings with the more general Norden–Rayleigh model (Norden 1960).

METHODOLOGY

Data sources and retrieval

For the proposed analysis of open source software development and its enactment in virtual communities, information on a large number of projects was necessary. Therefore, automated retrieval of public data (Cook et al. 1998) was performed, choosing SourceForge.net, the well-known and largest software development and hosting site, as the source of data. The mission of SourceForge.net is ‘to enrich the open source community by providing a centralized place for open source developers to control and manage open source software development’. To fulfill this mission goal, a variety of services is offered to hosted projects, including tools for managing support, mailing lists and discussion forums, web server space, shell services and compile farm, and source code control. While SourceForge.net publishes several statistics, e.g. on activity in their hosted projects, this information was not deemed detailed enough for this study. For example, Crowston and Scozzi (2002) used the available data for validating a theory for competency rallying, which suggests factors important for the success of a project. Hunt and Johnson (2002) have analysed the number of downloads of projects occurring, and Krishnamurthy (2002) used the available data of the 100 most active mature projects for an analysis.

To gather additional and more detailed information, data from the web pages and especially the source code control system in the form of CVS of the projects hosted were retrieved. CVS (Concurrent Versions System) is a free source code control and versioning system, which is being used extensively in the free software community (Fogel 1999), and whose aim is to coordinate a number of participants working together on source code, trying to maximize efficiency by allowing for concurrent work on checked out copies and providing for later merges if conflicts occur. The database stores each change to the code committed by a participant, thereby allowing reconstruction of prior states and comparisons between versions of source code files. In addition, meta-data on the work of the programmers within the project by submitting (‘checking in’, ‘committing’) files are stored.
Especially the changes in the lines-of-code, programmer name, file identification, date and further information are saved with each commit. Several works have already demonstrated that relevant information about both open source and commercial software development can be retrieved from such systems (Atkins et al. 1999; Mockus et al. 2000; 2002; Koch and Schneider 2002). For example, the changes in the lines-of-code stored with each commit, coupled with the programmer identification allow for analyses of participants’ contributions to a project overall and especially over time. Also people’s cooperation on source code can be derived from this data, and finally the progression of projects both in size and number of participants over time can be analysed and compared.

The retrieval process started with inspecting the SourceForge.net homepage for the published number of currently hosted projects (at the relevant date 23,000). All of these were selected as candidates for analysis. As not all projects are both still hosted and have the CVS service enabled, the CVS information web page hosted at SourceForge.net was queried for the information necessary for data retrieval, i.e. project and server name. This resulted in 21,355 candidate projects with enabled CVS service. In addition, the development status indicator for each project assigned by the project’s administrator was retrieved from its respective web page. Using the CVS web interface page provided for each project with enabled CVS service, 8,791 projects were identified which actively use this service. Together with the CVS server name information, this information was used to retrieve the necessary data from the CVS servers. This process was mostly managed by Perl scripts for web page querying and generating a shell script for CVS server access, in each case allowing ample sleeping periods so as not to delay services for other users. The output of each step was again parsed by Perl scripts for the relevant data, which were stored in a database. Analyses were performed by queries to this database and subsequent processing with R, a free language and environment for statistical computing and graphics. The following analyses are based on the 8,621 projects for which all relevant information could be retrieved. Projects which do not have CVS enabled, which are no longer hosted or which never have actively used the CVS repository are not included. There are several possible explanations for this, including the use of a similar service somewhere else, e.g. on a private server, no work on source code yet, or deletion because of complete migration to a different repository or again private server with application for deletion from Sourceforge.net, or even participants abusing the service or illegal activities. In any case, complete disuse of CVS or discontinuation on Sourceforge.net does not necessarily mean project failure, and therefore these projects can be not considered as such in the following and are completely discarded.

**Metrics used**

The first metric used is the number of lines-of-code (LOC) added to a file. The definition of this often-disputed metric (Humphrey 1995; Park 1992) is taken from the CVS-repository and therefore includes all types of LOC, e.g. also commentaries (Fogel 1999). In addition, any LOC changed is counted as one LOC added and one LOC deleted. The next metric is defined analogous and pertains to the LOC deleted. The difference between the LOC added and the LOC deleted therefore gives the change in size of a software artefact under consideration in the corresponding time period. These changes can be cumulated to give the size at any moment. The metric of commit refers to the submission of a single file by a single programmer. The total time spent on the project could be defined for every programmer as the difference between the date of his first and his last commit, but as this therefore includes all time elapsed, not necessarily only time spent actually working on the project, this measure would only give an upper bound for actual time spent working. Since Koch and Schneider (2002) have shown that this measure is not usable for predicting the output of a given programmer, their additional definition of a programmer as being active in a given period of time if he performed at least one commit during this interval will be adopted. As Koch and Schneider (2002) have demonstrated, this metric has a high correlation with the output produced, and is therefore usable for effort estimation. The last metric was directly taken from the SourceForge.net repository, which has a development status indicator assigned to each project by the project’s administrator. This indicator aims at reflecting the phase of a project in the development lifecycle and has seven possible values, reaching from planning, pre-alpha, alpha, beta to production/stable and mature, and to inactive. As its value is determined by the project’s administrator, it need therefore not necessarily be a correct description of the current status.

**PROJECT ECOLOGY**

For the 8,621 projects analysed, a total of 7,734,082 commits were made, with 663,801,121 LOCs added and 87,405,383 LOCs deleted. The projects consist of 2,474,175 single files, and an overall number of 12,395 distinct programmers have contributed with at least one commit.

The first important characteristic of the project ecology is the distribution of both the assets available, i.e. the programmers, and the resulting outcome, i.e. commits, LOCs and project status. Next, the relationships between these variables and their evolution over time need to be explored. Table 1 gives first descriptive statistics for these variables. As can easily be seen, mean and standard deviation do not give an adequate picture of the
distribution of most variables, except development status, which is ordinal-scaled. The other variables are clearly not normal-distributed (which can be ascertained using a Kolmogorov–Smirnov test).

Figure 1 shows the histogram of number of distinct programmers per project having committed at least one file to it. As can easily be seen, the distribution is heavily skewed, with a vast majority of projects having only a very small number of programmers (67.5% have only one programmer). Only 1.3% have more than 10 programmers. Analysing the 100 most active mature projects on Sourceforge.net, Krishnamurthy (2002) also showed that most of the projects had only a small number of participants (median of 4). Only 19% had more than 10, 22% only one developer. While this percentage is much smaller than found here, this is not surprising as Krishnamurthy only used the 100 most active projects, not all. Krishnamurthy also found positive correlations between number of developers and downloads of a project, and between age of the project and the number of developers. Hunt and Johnson (2002) have analysed the number of downloads of projects occurring. They show that the distribution of projects according to this number is also heavily skewed and follows a power law (or Pareto or Zipf) distribution. This form of distribution has been recognized in a number of fields including distribution of incomes, word usage and website popularities. A power-law implies that small occurrences are extremely common, whereas large instances are extremely rare. While there are several explanations for the occurrence of this sort of distribution, in the case of open source software development communities increased success, attractiveness and popularity with increasing number of programmers might lead to even more programmers, and thus constitute a positive feedback loop. To confirm whether a power-law relationship is present, a log-log-plot is drawn (see Figure 2). A line on this plot should indicate a power-law, and the resulting parameters from a linear regression using frequency as dependent and number of distinct participants as independent variable can be used. As can be seen, a linear regression results in an $R^2$ of 0.94 in this case, confirming the power-law.

Regarding the output of the projects, a similar situation can be seen (see for example Figure 3). The vast majority of projects achieve only a small number of commits and are of small size. The development status is more evenly distributed, with nearly the same proportion for each possible status (see Figure 4). But it should be noted that this indicator is assigned by the project

### Table 1. Descriptive statistics for project variables ($n=8,621$)

<table>
<thead>
<tr>
<th>Variable</th>
<th>Min.</th>
<th>Max.</th>
<th>Mean</th>
<th>Std. deviation</th>
<th>Median</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of programmers</td>
<td>1</td>
<td>88</td>
<td>1.86</td>
<td>2.61</td>
<td>1</td>
</tr>
<tr>
<td>Commits</td>
<td>1</td>
<td>133,759</td>
<td>897.12</td>
<td>3,840.90</td>
<td>192</td>
</tr>
<tr>
<td>LOC added</td>
<td>0</td>
<td>12,951,218</td>
<td>76,998.16</td>
<td>458,975.28</td>
<td>10,801</td>
</tr>
<tr>
<td>LOC deleted</td>
<td>0</td>
<td>3,846,863</td>
<td>10,138.66</td>
<td>73,493.03</td>
<td>373</td>
</tr>
<tr>
<td>Files</td>
<td>1</td>
<td>42,674</td>
<td>285.46</td>
<td>1,317.74</td>
<td>69</td>
</tr>
<tr>
<td>Development status</td>
<td>0</td>
<td>6</td>
<td>2.67</td>
<td>1.77</td>
<td>3</td>
</tr>
</tbody>
</table>

Figure 1. Histogram of distinct programmers per project

Figure 2. Log-log-plot of distinct programmers per project and frequency with line resulting from linear regression
These results lead naturally to the assumption that input and output of projects are correlated, i.e. that projects with a small number of programmers only achieve small numbers of commits and LOCs. This intuitive relationship is indeed ascertained. At a significance level of 1%, the total number of programmers of a project correlates positively with the output measures, with coefficients 0.472 with number of commits, 0.408 with total LOCs added, and 0.192 with development status (in this case using Spearman’s coefficient). Regarding the starting time of the project, i.e. the date of the first commit, there is a significant negative correlation of about −0.2 with the output measures and −0.197 with number of programmers, confirming Krishnamurthy’s findings (2002). The output measures of commits and LOCs added correlate highly at 0.872, the development status with commits and LOCs both at 0.288. These results confirm the intuitive notion that attraction of a large number of programmers is vital to the success of an open source project, but the effects on productivity need to be further explored.

Another interesting aspect to explore is the evolution of open source projects. The study of software evolution was pioneered by the work of Belady and Lehman (1976) on the releases of the OS/360 operating system, which has led to many other works (e.g. Lehman and Ramil 2001), in which the laws of software evolution were formulated, expanded and revised. These laws entail a continual need for adaptation of a system, followed by increased complexity and therefore, by applying constant incremental effort, a decline in the average incremental growth. Turski has modelled this as an inverse square growth rate (Turski 1996). The first study on software evolution in open source systems was performed by Godfrey and Tu (2000), who analysed the Linux operating system kernel and found a super-linear growth rate, contradicting the prior theory of software evolution, which would give an indication of major differences in development modes and their results. The SourceForge.net project ecology offers a multitude of projects to validate this theory. As a first approach to be detailed here, both a linear and a quadratic model were computed for each project, taking the size in lines-of-code as a function of the time in days since the first commit, which is used as project start date, and using one month as time window (Godfrey and Tu 2000). This was possible for 4,047 of the projects, the others lacked necessary information (i.e. had too few data points). The quadratic model, not surprisingly as it is more flexible, outperformed the linear one with a mean $R^2$ of 0.923 (median 0.954) against 0.786 (median 0.832). Using a Wilcoxon signed rank test instead of a paired-samples t-test because the difference scores are not normal distributed, the hypotheses that the distributions are equal is indeed rejected (at 1%). The most interesting fact to explore is whether or not the growth rate decreases over time according to the laws of software evolution. This can be checked by analysing the second derivative of the quadratic model (or directly using the coefficient of the quadratic term), which is confirmed to be different from 0 (using a t-test at 1%). The distribution of this quadratic term has a slightly negative mean of −0.504 (median −0.020), indicating decreasing growth rate in accordance to the laws of software evolution, but contrary to the findings of Godfrey and Tu (2000) for Linux. In fact, for 61% of the projects this term is negative, for the remaining minority (1,578) positive, therefore showing also a rather large number of projects exhibiting super-linear growth. If only projects having achieved a given maturity (status productive or mature) are included in the analysis, which leaves about a quarter (1,087 projects) of the data, the results stay the same. To explore whether there are
any characteristics of projects that lead to super-linear growth, the projects are divided into two groups according to their growth behaviour (super-linear or not). Using (Spearman) correlations and Mann-Whitney U-tests a small (but significant) relationship with size (in lines-of-code) and number of programmers can be found, indicating that larger projects with a higher number of participants might be more often able to sustain super-linear growth. This would be in accordance to the findings of Godfrey and Tu (2000), as Linux is a relatively large project, but would contradict the assumption of software evolution that increased size leads to more complexity and interdependencies, thus decreasing growth rate. Open source software development, through measures like strict modularization and self-selection for tasks seems to be able to at least delay this effect (a notion which is further explored together with productivity). In addition, the fourth law of software evolution, ‘conservation of organizational stability’ (Lehman and Ramil 2001), implying constant incremental effort, might be violated especially in large projects which attract an ever increasing number of participants.

PARTICIPANTS AND THEIR CONTRIBUTION

Most previous studies have found a distinctly skewed distribution of effort between participants in open source projects. For example, Mockus et al. (2002) have shown that the top 15 of nearly 400 programmers in the Apache project added 88% of the total lines-of-code. In the GNOME project, the top 15 out of 301 programmers were only responsible for 48%, while the top 52 persons were necessary to reach 80% (Koch and Schneider 2002), with clustering hinting at the existence of a still smaller group of 11 programmers within this larger group. A similar distribution for the lines-of-code contributed to the project was found in a community of Linux kernel developers by Hertel et al. (2003). Also the results of the Orbiten Free Software survey (Ghosh and Prakash 2000) are similar, the first decile of programmers were responsible for 72%, and the second for 9% of the total code.

Once again, similar results can be found at the project ecology under consideration. Table 2 shows first descriptive statistics for the variables describing single programmer’s contribution to the project ecology. As can also be seen from the graphical representations (see Fig 5 and 6 for examples), the distributions again show a vast majority with small contributions and some high-performers. A Kolmogorov–Smirnov test again

<table>
<thead>
<tr>
<th>Variable</th>
<th>Min.</th>
<th>Max.</th>
<th>Mean</th>
<th>Std. deviation</th>
<th>Median</th>
</tr>
</thead>
<tbody>
<tr>
<td>Commits</td>
<td>1</td>
<td>132,736</td>
<td>624</td>
<td>2,815</td>
<td>112</td>
</tr>
<tr>
<td>LOC added</td>
<td>0</td>
<td>16,152,866</td>
<td>53,554</td>
<td>374,750</td>
<td>5,469</td>
</tr>
<tr>
<td>LOC deleted</td>
<td>0</td>
<td>3,846,863</td>
<td>7,052</td>
<td>54,547</td>
<td>372</td>
</tr>
<tr>
<td>Files</td>
<td>1</td>
<td>44,271</td>
<td>230</td>
<td>1,105</td>
<td>48</td>
</tr>
<tr>
<td>Projects</td>
<td>1</td>
<td>15</td>
<td>1.29</td>
<td>0.81</td>
<td>1</td>
</tr>
</tbody>
</table>
not surprisingly rejects normal-distribution. To allow for comparison with the results above, Figure 7 shows responsibility for the total source code. As can be seen, the top decile is responsible for 79%, with the second decile responsible for an additional 11%. These results, while spanning several projects, are clearly in accordance with prior findings. Any consequences for productivity within single projects this form of distribution might have will be explored later.

The small number of projects that programmers work on seems quite interesting, while being in accordance with prior findings. Ghosh and Prakash (2000) have also found that most of the programmers, exactly 11,500 out of the 12,700 analysed, have only worked on one or two projects. In the ecology considered here, an even greater amount of 94.1% worked on less than three projects. This seems interesting, as Ghosh and Prakash were reviewing a Linux distribution and freshmeat projects, while the ecology analysed here constitutes a single hosting and entry point for all projects. The existence of such a point and the resulting collocation of projects does therefore, maybe counter-intuitively, not lead to increased participation in other projects on the same hosting site. As the information of which programmers work on which projects is available, further analyses might include author clustering as proposed by Ghosh (2003), building a graph consisting of projects as vertices and edges representing common participants (Madey et al. 2002).

Not surprisingly, analysing the relationships between these variables of programmers’ contribution yields high positive correlations (all significant at 1%). For example, the number of commits and lines-of-code added have a coefficient of 0.902, commits and number of files worked on of 0.975, and the number of different projects has coefficients of about 0.3 with the other measures. The time of first commit at the ecology overall also correlates at about −0.3 with the other variables, which is not surprising as a programmer having joined earlier will generally have performed more work. As a further metric, the mean number of lines-of-code added per single commit was computed, to uncover potential differences in working style. There is indeed a significant positive correlation with several other measures, e.g. of 0.228 with number of commits, 0.582 with total lines-of-code added and 0.252 with number of files worked on. This seems to indicate that programmers with higher contribution use larger commits, and therefore exhibit a different style at least in this regard. Similar to the results of Koch and Schneider (2002) only a small number of programmers cooperate on the basis of single files (mean of 1.16 programmers work on a file, median 1). If the contribution of programmers over time is analysed (using one month as period), it can be seen that the large differences in total contributions are mostly due to different intensities of contribution, not longer time. The mean number of commits and lines-of-code added in the active months of a programmer, i.e. those with at least one commit, are highly correlated with the respective total numbers (at 0.895 and 0.922), and show the same distribution within the population.

**PRODUCTIVITY AND EFFORT**

**Effects on productivity**

There are several factors in a project that might influence the productivity within the project. As a first idea, the distribution of effort in the development team is explored. As detailed above, the distribution is heavily skewed, with only a small minority being responsible for most of the output. The question to be answered is whether this is a good way of organizing the work, i.e. does this form of distribution leads to good results? As a measure of the inequality of work distribution within the development team, for each programmer in the team, the squared difference between the percentage of commits he contributed, and the percentage he should contribute if everyone contributed the same amount, was computed (excluding single-person projects). For example, programmer X might be responsible for 80% of the commits in a four person-team, while he should be responsible for 25% in a team of equals. Then the difference between the percentage of commits he contributed, and the percentage he should contribute if everyone contributed the same amount, was computed (excluding single-person projects). For example, programmer X might be responsible for 80% of the commits in a four person-team, while he should be responsible for 25% in a team of equals. Then the difference between the percentage of commits he contributed, and the percentage he should contribute if everyone contributed the same amount, was computed (excluding single-person projects). For example, programmer X might be responsible for 80% of the commits in a four person-team, while he should be responsible for 25% in a team of equals. Then the difference between the percentage of commits he contributed, and the percentage he should contribute if everyone contributed the same amount, was computed (excluding single-person projects).
number of programmers and the development status, while positive and significant, are below 10%. Interestingly, there is no correlation with the age of the project, so the inequality does not increase simply with time. These results do not change significantly if the inequality is measured based on lines-of-code instead of number of commits. Of course, the results given do not necessarily indicate that more activity in projects is caused by a more unequal distribution of contributions, as the other way would also give a possible explanation, i.e. as the project grows, the inequality grows as a result.

The next possible influence on productivity in a project is the number of active programmers. Following the reasoning of Brooks, an increased number of people working together will decrease productivity due to exponentially increasing communication costs (Brooks 1995). Therefore, the number of programmers and the achieved progress in each project was analysed on a monthly basis. The first interesting result was, that although the number of active programmers has a significant and positive relationship with the number of commits and lines-of-code in a given period, the coefficient is much smaller than previously found by Koch and Schneider (2002) in their analysis of the GNOME project. They found a correlation of 0.932 between active programmers and number of lines-of-code added, while in the project ecology considered here (again only considering projects with more than one participant) the coefficients are only 0.072 for lines-of-code added and 0.194, slightly higher, for number of commits. To uncover the effect an increased number of people working together has on productivity, the relationship to the mean number of commits and lines-of-code added per programmer in a period is explored. According to theory, the output per person should decrease if the number of persons increases. But surprisingly, this effect is next to non-existent, as the correlation coefficient is (although negative) only -0.013 with both measures (significant at 5%). This leads to the interesting conclusion that Brooks’s Law seemingly does not apply to open source software development. There are several possible explanations for this, which include the very strict modularization, which increases possible division of labour while reducing the need for communication. Also the low number of programmers working together on single files can be taken as a hint for this.

**Effort estimation**

The main indicator for how the open source software development model compares with the traditional, commercial one is the effort expended. As not even project leaders know how much time is expended by their participants, as no time sheets or similar mechanisms are employed, this effort for the software development needs to be estimated. For this task of effort estimation, literature yields several methods that have been developed in the context of commercial software development. These include the well-known COCOMO (Boehm 1981), which offers, depending on the selection of one out of three development modes, an algorithmic formula for estimating the effort based on a quantification of the lines-of-code. This model has lately been modified and updated with the publication of COCOMO II (Boehm et al. 2000). Other options for effort estimation include the software equation by Putnam (1978), function points (Albrecht and Gaffney 1983), diverse machine-learning approaches or proprietary models like SLIM or ESTIMACS. Many of these models are based on a general formulation of a development project introduced by Norden (1960), termed the Norden–Rayleigh model.

Establishing effort estimation for open source software development has an additional positive effect besides allowing for comparisons with other development paradigms — while the results of an effort estimation in commercial development are used for planning and control by management, there are also stakeholders in open source projects who could be interested in such results at early stages or during a project. These include the community itself, especially, dependent on the organizational form, the owner/maintainer, inner circle or committee (Raymond 1999), which need to monitor progress and plan for release dates, and programmers considering whether to join or to remain in a project. Further possible interested parties are current or prospective users, who need the functionality at a given date or with a given maturity level, and especially corporations which are intending to pursue a business model based on this software, need it for their operations or plan to incorporate it in their products or services.

The problems faced when estimating the effort for open source projects, in addition to the usual problems inherent in effort estimation, include the potential high turnover of personnel, reduced productivity due to larger number of participants (a point which has already been disproved above), and the voluntariness of people’s participation. The last point has also been addressed by Koch and Schneider (2002), who have shown that at least in the GNOME project the staffing surprisingly closely follows the Norden–Rayleigh model (Norden 1960, Putnam 1978) proposed decades ago. In addition, several assumptions of effort estimation models are inherently violated in open source development. For example COCOMO (Boehm 1981) assumes development following a waterfall model and permanence of the requirements during the whole process. As the requirements are neither written down (Vixie 1999) nor constant over time, and the software development follows a more spiral type of approach, having been termed micro-spirals (Bollinger et al. 1999), both assumptions are violated. Nevertheless, Wheeler (2002) used the basic model of COCOMO in organic development mode on the Red
follows this model, also reaching peak manning at the function, i.e. the number of active developers, closely model for the GNOME project, in which the manpower expended can be computed. Koch and Schneider (2002) for the Rayleigh-curve and the total manpower to be of peak manning has been reached, both the parameter determination of the peak manpower. If the point function therefore represents a Rayleigh-type curve to the exhaustion of the problem space. The manpower deployment of the software, and then decreases due to increasing and decreasing economies of scale, a prototype-oriented software process and flexibility in the requirements. Koch (2003) has shown that possible parameter values can indeed be found in this model to reproduce the Norden–Rayleigh estimation of the GNOME project. Therefore COCOMO II is employed next, using realistic values for the necessary parameters, arriving at a total effort of 135,000 person-years, with mean 15.7 and median 1.69 for single projects. As this model allows for a wider range of development models, it is possible to capture adequately the efficiency Koch (2003) found in the GNOME project, therefore the resulting effort is lower than in the previous version.

As a next and most general approach, the Norden–Rayleigh model will be explored (Norden 1960). Starting from the main idea that any development project is composed of a set of problems that need to be solved by the manpower employed, the application of which is governed by a learning rate linear in time, the number of people usefully employed at any given time is assumed to be approximately proportional to the number of problems ready for solution at that time. Therefore, the manpower function increases until the point of peak manning, which as Putnam (1978) has shown is close to the deployment of the software, and then decreases due to the exhaustion of the problem space. The manpower function therefore represents a Rayleigh-type curve governed by a parameter that plays an important role in the determination of the peak manpower. If the point of peak manning has been reached, both the parameter for the Rayleigh-curve and the total manpower to be expended can be computed. Koch and Schneider (2002) and Koch (2003) have demonstrated the use of this model for the GNOME project, in which the manpower function, i.e. the number of active developers, closely follows this model, also reaching peak manning at the time of first major release. This in-depth analysis including consideration of release dates is impossible for a whole project ecology with more than 8,000 projects. Therefore the point in time of peak manning and the respective number of active programmers are retrieved for each project. From this, the Rayleigh-curve is constructed automatically, and the total manpower to be expended is computed. This manpower is computed in person-years, but as the number of active open source programmers forms the basis, also open source programmer-years result (which normally are less than commercial programmer-years based on a 40 hours week). Using a mean number of working hours of open source programmers, this measure can be converted (Koch and Schneider 2002, Koch 2003). For example, Hertel et al. (2003) report that in the developer group of Linux kernel contributors participating in their survey, about 18.4 hours per week are spent on open source development by each person. As the distribution of effort within their group is similar to the one found in this project ecology, this measure could be taken. The results are striking nevertheless: The sum of efforts for all projects resulting from Norden-Rayleigh-modeling is 12,967 open source programmer-years, with mean effort for a project of 1.5 and median 0.41 open source programmer-years. Not surprisingly, the resulting distribution is once again heavily skewed. Using conversion to commercial programmer-years, the resulting sum of efforts is only 5,965 person-years, considerably less than the results achieved using COCOMO and COCOMO II (about 4.4% of the COCOMO II estimation). As this approach to estimating the effort is based on actual participation information, not only on lines-of-code like in COCOMO, it seems nearer to the truth. Indeed, these results can be taken as a hint that the efficiency of open source software development is very high, as the results derived from actual participation are considerably smaller than the ones derived from the resulting product. This means that the input is smaller than would be expected in the commercial world regarding the output. There are several possible explanations, including the absence of management overhead, the seeming absence of productivity losses due to increased participation and the self-selection for suitable tasks which seem to lead to highly efficient self-organizing online communities.

There is an additional methodological difference between the COCOMO and the Norden–Rayleigh approach: using COCOMO, each project is seen as a project having finished with the respective size in lines-of-code, while Norden–Rayleigh gives the total effort needed to finish the software, including maintenance and enhancements, given that the peak manning seen as yet is the peak manning for the whole project (and also the time of first operation). Therefore both will underestimate the effort for projects not progressed very far yet, COCOMO because it does not assume any further increase in lines-of-code, Norden–Rayleigh because the peak manning would not have been reached. Using only
phenomena and differences with existing theory from commercial contexts encountered in open source software development, basing any assessment not only on a small number of well-known and successful projects, but on a complete and heterogeneous population.

As the results of this work show, there are indeed significant differences between this cooperative development model and the commercial organization of work. Open source software projects differ significantly in their output, i.e. the achieved size and activity, and their ability to gather input, i.e. a large number of developers, forming an example for a power-law relationship. The evolution of projects’ size over time seems in part to contradict the laws of software evolution proposed for commercial software systems. While a majority follows these laws and exhibit a decreasing growth rate, about one third of the projects are able to sustain super-linear growth. Interestingly, these projects tend to be larger and are based on the cooperation of more participants, forming a distinct contradiction to prior theory.

The programmers themselves, as found in other studies, differ significantly in their contributions within the project ecology, with a small group being responsible for a large part of the output. This difference seems mostly to come from sustained higher intensity of participation, not necessarily longer duration. Interestingly, the number of projects that developers work on is very small, similar to the number found in prior studies, leading to the conclusion that collocation of projects in a virtual hosting community does not increase co-participation.

The implications of these results on productivity within projects has shown some surprising results: The inequality of effort distribution between programmers within a project increases with the size of the project in number of commits and lines-of-code, but not with age. Contrary to software engineering wisdom, an increasing number of active developers does not lead to a decrease in productivity. This is a major confirmation for the validity of this development model and the reasoning of its advocates that attraction of a large number of participants will be beneficial.

Lastly, the effort expended in the open source online community considered here was explored, based on several estimation models. The results from COCOMO, which seems problematic but is applied for conformity with other studies, and the newer COCOMO II, allowing for more possibilities of capturing modern development paradigms, but both relying on the produced source code, respectively showed a large amount of 160,000 and 135,000 person-years expended. Using for each project an automatically estimated Norden–Rayleigh model, which relies on data of people’s participation only, showed a distinctly smaller expenditure of 86,000 person-years. This might lead to two conclusions — either the open source software development model is very efficient in its relation between effort and software produced, or the people participating in tasks
other than writing code account for the vast majority of expended effort, thus forming an extreme example of ‘chief programmer’ or ‘surgical’ teams.

In summary, a multitude of findings is possible from analysing an online community of open source projects and programmers. The general trend of the results presented here shows striking differences between the open source development model and prior assumptions and theories. Especially the results of effort estimation propose that either a highly efficient way of building software is achieved by self-organizing cooperative and highly decentralized work, or that the participation of users besides direct programming tasks is enormous and constitutes an economic factor of large proportions. This would also be an interesting facet to explore in other settings, as user-led development and higher levels of consumer participation, e.g. in ‘private-collective’ innovation models (von Hippel and von Krogh 2003) or by providing innovation toolkits (Franke and von Hippel 2003) might be an option for other areas besides software as well.

References


