Free/Libre and Open Source Software:

Survey and Study

FLOSS

Deliverable D18: FINAL REPORT

Part V: Software Source Code Survey

Rishab Aiyer Ghosh

Gregorio Robles

Ruediger Glott

International Institute of Infonomics

University of Maastricht, The Netherlands

June 2002

The original version of this document is available at

http://www.infonomics.nl/FLOSS/report/

Table of Contents

List of Figures

1.1. Free software developers: a starting point for measurement

In the past two years there have been some surveys conducted, of users as well as developers, though usually on fairly small samples and far from comprehensive. No survey actually looks at what is perhaps the best source of information on free software – the source code itself. This was attempted first as an experiment in late 1998 and then published after more work as the Orbiten Free Software Survey in May 2000[1]. Although there have since been other surveys of authorship[2] and many of the relatively recent web sites that provide an environment for open source development such as SourceForge provide some statistics, none of these adopt the approach of looking at the free software community from the bottom up. With the result that simple facts, such as the number of individual developers contributing to free software projects, an indicative number of such projects and their size were unknown.

1.1.1. How software tells its own story

The Orbiten Survey took advantage of one of the key features of the software development community. In contrast to other "cooking pot markets" on the Internet such as newsgroups and discussion forums, much of the activity around is precisely recorded. The "product" – software – is by nature archived. Since source code is available, the product is open to scrutiny not just by developers, but also by economists. Arguably all economic activity: production, consumption and trade – in the Internet's cooking-pot markets is all clearly documented, as it is by nature in a medium where everything can be – and much indeed is – stored in archives.

The difference between software and discussion groups – where too the "product", online discussions, is available in archives – is that software is *structured*. To understand what is going on in a discussion group, one might need to read the discussions, which is quite complicated to do in an automated fashion. However, reading and understanding software source code is by definition something that is very easily done by a software application.

Software source code consists of at least three aspects that are useful for economic study. It contains *documentation* – the least structured of all the data here, since it is written in a natural language such as (usually) English. This provides

[1] Ghosh & Ved Prakash, 2000
[2] WIDI 2000; Jones 2002

information on among other things the authorship of the software. *Headers* are called different things in different programming languages but perform the same function, of stating dependencies between the software package under scrutiny and other software packages. Finally, the *code* itself provides information on the function of the software package. As an automated interpretation of this is exactly what happens when the program is compiled or run, there may be far too much information there to be usefully interpreted for an economist's purpose. But it is possible to have an idea of the importance or application domain of the code in some subjective (if well-defined) sense – it works with the network, say, or has something to do with displaying images.

Naturally these categories are not sharply divided – indeed most authorship information for individual components of a software package may be present through comments in the code, which fits, for current purposes, the category of documentation.

There are formalized procedures for authors to declare authorship for entire packages on certain repositories and archives, but such information needs to be treated carefully too. The data may be reliably present, but its semantics are variable. Usually such "lead authors" hold responsibility for coordination, maintenance and relations with a given repository, but data on other collaborating authors – let alone authorship of individual components – may be entirely missing. On the other hand such detailed data are usually present in the source code itself.

1.1.2. What may be inferred

There is little point doing a small "representative" survey since results are meaningless unless very large amounts of software are processed. Given the data at hand, and the degree of structural complexity for automation – there is a cornucopia of interesting findings to be made. At the very simplest, a map of author contribution can be made, resulting in an indicator of the distribution of non-monetary "wealth" or at any rate production. This is in theory simple to do – count the lines of code and attribute that figure to the author(s) with the nearest claim of credit.

More complicated is to look for links between projects and groups of projects, as well as links between groups of authors. The former can be done by looking for dependencies in the source code – references from each software package to other software packages. The latter is inferred through the identification of authors who work on the same project or group of projects. Of course both these indicators refer to one another – projects with related authors are in some way related projects; authors of a project that depends on another project are in a way dependent on that other project's authors.

Measuring such dependencies and interrelationships can provide an insight into the tremendous and constant trade that goes on in the free software cooking-pot markets, and can probably also provide an indicator of the relationship with commercial software and the (monetary) economy at large. Finally, the value of all such parameters can be applied over the fourth dimension, either using a simple chronology of time, or the virtual chronology of multiple versions of software packages, each of which replaces and replenishes itself wholly or in part as often as every few weeks.

1.2. What is in the source: extracting data from source code

We proceed to look further into the details and format of empirical data that can be extracted through a primarily automated scan of software source code. The degree (and reliability) of extractability, as it were, depends on the type of data extracted. These fall into four broad categories.

- Authorship information for source at the sub-package/component level
- Size and integrity information for source code at the package level[3]
- The degree of code dependency between packages

All these data can also be collected chronologically, i.e. over different versions of source code or of source packages at different points in time.

1.2.1.Authorship information

Authorship information is perhaps the most interesting yet least reliable of the data categories. Although most FOSS developers consider marking source code they've written as important[4] they apparently do not take sufficient care to do so in a consistent manner. Claiming credit is usually done in an unstructured form, in natural-language comments within source code, posing all the problems of automated analysis

[3] a package, loosely defined, is several files distributed together. Usually a package can be reliably dated to a specific version or release date. Sub-packages are the individual files or collections of files at the next lower level(s) of the distribution directory structure

[4] According to the FLOSS developer survey, 57.8% consider it "very important" and a further 35.8% don't consider it "very important" but claim to mark their code with their names anyway; see http://floss1.infonomics.nl/stats.php?id=31

of documentation. Several heuristics have been used, however, to minimise inaccuracies and are described further in the technical documentation for the software scanning application CODD[5].

Particular issues or biases that have not yet been fully resolved include several cases of “uncredited” source code. This is either a result of carelessness on the part of authors, or in some cases, a matter of policy. Developers of the web server Apache[6], for instance, do not sign their names individually in source code. A large amount of important source code is the copyright of the Free Software Foundation, with no individual authorship data available[7]. However, these specific situations do not affect the integrity of the data in general. Indeed, in general this method of determining authorship by examining the source code itself shares (some of) the bias of alternative methods towards crediting lead authors, as many authors who contribute small changes here and there do not claim credit at all, handing the credit by default to lead authors.

On the other hand, this bias is possibly balanced by a bias introduced towards the other side by the CODD heuristics, which usually give equal credit to multiple authors when they are listed together with no identifiable ranking information (thus narrowing the difference between a lead author and a minor author in case they are listed jointly).

Alternative methods

There are alternative methods of assessing authorship of free/open source software. Typically, they are based on more formal methods of claiming credit. In the Linux Software Map, for example, it is usually a single developer who assumes the responsibility for an entire package or collection of packages that are submitted to an archive. On collaborative development platforms such as SourceForge, similar methods are used; specific authors start projects and maintain responsibility for them. With these methods, assessing authorship is limited to collating a list of “responsible” authors. Clearly the semantics of authorship here are quite different from what we

[5] Designed by Rishab Ghosh and Vipul Ved Prakash, and implemented by Vipul Ved Prakash. The first version of CODD was created in 1998 and the name was an acronym, the expansion of which we cannot recall, though it was possibly “Concentration of Developer Distribution”. See also http://orbiten.org/codd/

[6] www.apache.org

[7] Several authors formally assigned their copyright to the FSF in order to protect themselves from liability and increase the enforceability of copyright. Assignment records are not yet available for access to academic research.

have previously described, since "responsible" authors may be responsible for maintenance without actually authoring anything, and in any case there are several contributors who are left out of the formal lists altogether. Thus, any attempt at identifying clusters of authors is likely to fail or suffer considerable bias.

A more detailed and less biased (but also less formal) method of author attribution is used by developers themselves during the development process. Either through a version-control system, such as CVS or Bitkeeper[8], or simply through a plain-text "ChangeLog" file, changes are recorded between progressive versions of a software application. Each change is noted, usually with some identification of the person making the change – in the case of a version control system this identification, together with the date, time and size of change is more or less automatically recorded. However, again the semantics vary – most projects limit to a small number the people who can actually "commit" changes, and it is their names that are recorded, while the names of the actual authors of such changes may or may not be.

Naturally, no method is perfect, but the purpose of the above summary is to show that formal author identification methods do not necessarily provide much additional clarity into the nature of collaborative authorship, while introducing their own biases. (However, CODD is being adapted to process CVS/Bitkeeper records as well.)

1.2.2.Size and integrity

There are many ways to value the degree of production a specific software package represents. Especially when it does not have a price set on it, the method of choosing an attribute of value can be complex. One value, which makes up in its completely precise, factual nature what it may lack in interpretability is size. The size of source code, measured simply in bytes or number of lines, is the only absolute measure possible in the current state of F/OSS organisation and distribution. Specifically, measuring the size of a package, and the size of individual contributions, allows something to be said about the relative contributions of individual authors to a package, and of the package to the entire source code base. It may also be possible to impute time spent in development or some a monetary value based on size, although we do not attempt to do so.

[8] CVS: Concurrent Versions System, http://www.cvshome.org; Bitkeeper: http://www.bitkeeper.com

In any case, in order to calculate the size of a package it is important to try to ensure its integrity. A given package – especially on development platforms – usually includes derivative or "borrowed" works that have been written separately by other developers, but may be required in order to for the package to run. These are not necessarily identified as "borrowed" and could, in theory, be counted twice. Furthermore, they can artificially inflate the apparent contribution of an author of a "borrowed" work. CODD tries to resolve this by identifying duplicate components across the entire scanned code base and allocating them to only a single package wherever possible. This promotes integrity and avoids double-counting, and also provides information useful for finding dependencies between packages, by replacing "borrowed" works with external references to those works.

1.2.3.Code dependency between packages

Since software is by nature collaborative in functioning, software packages usually depend on features and components from several other packages. Such dependencies must be explicitly detailed in a way that they can be determined automatically, in order for an application to run. As such, these dependencies can be identified through automatic scanning; indeed there are several developers' tools that serve this purpose. Such tools normally provide (of necessity) a high level of detail regarding dependencies (i.e. at a function call level) well beyond the present purposes of analysis. Author credit information is rarely available at anything more detailed than file level, so it makes little sense to determine dependency information at a more detailed level. Moreover, such detailed analysis would be computationally exceptionally hard to perform for 30,000 software packages!

It was decided therefore to implement original but (relatively) uncomplicated heuristics to identify dependencies at the package level. One method is to retain information on duplicate files and interpret that as dependency information: if package P contains a file that has been "borrowed" from package Q where it originally belongs, P is dependent on Q.

Another method is based on header files. As described earlier, headers (called different things in different programming languages) define interfaces to functions, the implementations of which are themselves embodied in code files[9]. In order to access externally defined functions, a code file must typically *include*[10] a declaration for it, typically in the form of a statement including a header file. This is treated by CODD as an external reference. Various heuristics are used to identify, wherever possibly, the package where header file functions are actually implemented, and external references are resolved, wherever possible, as links from one package to another.

This process is quite complex given the unstructured nature of a large source base containing several possibly incompatible packages (i.e. which have not been designed to be installed together or run on the same system). Nevertheless, it can scale to a very large code base (tested so far on 30 gigabytes of software, over 22,000 packages), and it results in a relatively coherent map of code dependencies between individual packages. In the current stage of analysis, however, tabular dependency information is not converted into the format of a graph, although that would make analysis of clusters of dependency easier. Arguably a small package that is required by several others is more valuable than a large package without dependents, so further analysis of dependency information is very useful in order better to gauge the value distribution of packages. Moreover, it is possible to identify clusters of projects based on their interdependence, in addition to the clusters of projects based on common authorship.

A summary of the stages described so far is presented in table 2 below.

Method	Explanation	Resulting data
Authorship credits	Heuristics for determining and assigning authorship of code segments at the file or package level.	List of the form {author, contribution in bytes of code} generated for each package
Duplicate file resolution	Many files are included in several packages, intentionally or by mistake. This results in double counting (a file is credited to its author multiple times, ones for each package where it occurs). Various heuristics are used to try and resolve this problem	Corrected version of authorship credit list. List of shared files for each package.
Dependency identification	Files in one package may link to files in other packages. Heuristics are used to try and identify these links and also	For each package, a list of linked or "borrowed" files together with the names of their

[9] For the C/C++ programming languages, which amount for the largest proportion of general-purpose F/OSS, files ending with ".h" or ".hpp" are headers and those with ".c" or ".cpp" contain implementation code.

[10] Using the #include command in C/C++ source code, and other methods in other programming languages.

	identify where possible, in the case of duplicate files, which is the “owner” package and which is the “dependent” one	“owner” packages based on identifiable information.

Table 2: Summary of stages of source code analysis and resulting data format

1.3. Conclusion and practical considerations

This paper has so far described in some detail a proposed methodology to extract and interpret empirical data out of software source code The first large-scale application of the more sophisticated methods is presented below, with some conclusions and practical considerations based on a preliminary analysis of the application of this methodology on a large scale.

1.3.1. State of current data and tools

Existing data is a result of running the various tools described above on a very large base of software, 30 Gigabytes of compressed source code, i.e. approximately 3 billion lines. Partly due to the scale of this code base, the analysis is carried at a fairly high level in that packages are rather large and not broken down into smaller sub-packages (the Linux kernel, for instance, is treated as a single – albeit large – package, which means that dependencies or clusters are not identified for kernel components or sub-packages). Additionally, existing data is for current available versions without any historical data or chronological analysis.

Current analysis tools are entirely non-interactive software and fairly technical – i.e. they are not user-friendly to operate and need programmer skills for customisation tasks. The analysis does not provide graphical or visualization output, and there are at present no software tools as part of this project that perform chronological analysis. However, the development of such tools may not be necessary if it turns out that analysis of historical trends, say, is practical with the application of standard statistical analysis packages to data as currently generated. So far, this has seemed impractical – the difficulty of dealing with a graph of over 23,000 projects and 36,000 authors in a statistical package was one of the initial reason to develop customised methods and tools.

A preliminary evaluation of the methodology in practice must, however, be positive. It is perhaps unsurprising (but previously impossible to prove) that F/OSS projects are highly interconnected, with large amounts of code dependency and reuse.

It will take some experimentation, together perhaps with visualisation techniques, to tailor the tools to generate clusters of manageable sizes that can be compared with one another as distinct groupings. This is essential in order to make full use of the available data, by integrating the code dependency information with the clusters of authorship to determine the dependencies between distinct groups of authors on one another. However, this is beyond the scope of the FLOSS project.

If performed over multiple versions or over time, this analysis provides extremely interesting information on the “trade” between groups, and could be a first step towards determining the internal economics of the functioning of F/OSS development.

To conclude, these methods are a first attempt to provide concrete empirical data and analysis based on the source code – the only hard fact of F/OSS – and extract the most of what is already ubiquitous, waiting to be studied. Empirical data extraction from source code should be of great interest to economists and social scientists – but is also a valuable tool for developers to know about themselves and their organisation. This perhaps explains F/OSS developers’ continuing interest in CODD and the Orbiten survey[11].

[11] The first CODD results in end 1998 received several hundred thousand hits in a few days, as did the first Orbiten Free Software Survey in May 2000. These only provided author contribution tables, and for a very small source code base.

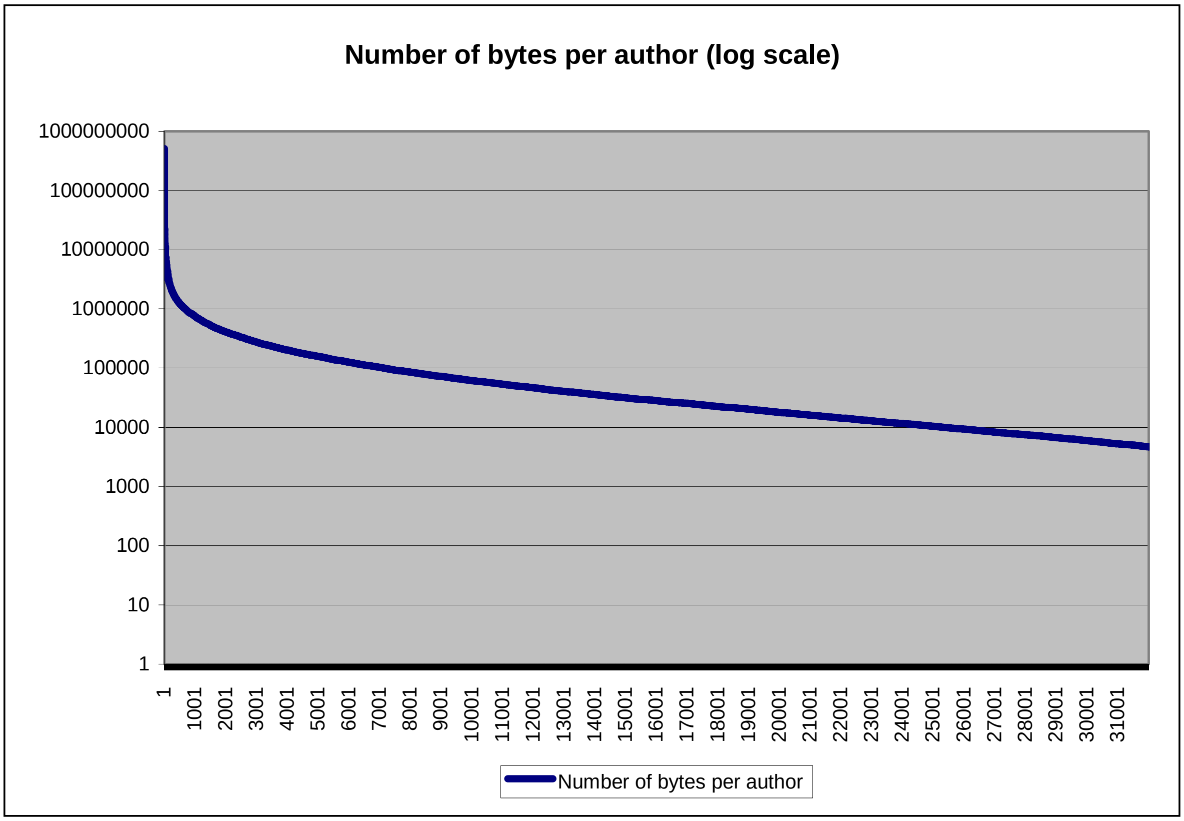
1.4. Authors' Contribution to OS/FS Projects

1.4.1. Authored Code

The purpose of this step of the analysis was to find out how the input to OS/FS projects, i.e. the number of program lines or bytes of source code, is structured. Based on the CODD-analysis, we present here the data for 31999 software developers collaboratively developing almost five billion (4.976.559.414) bytes of software source code.

Figure 1 shows the contributions of the OS/FS authors to the total sum of analysed software source code, where the order of the authors is ranked by the size of their contribution. It becomes clearly visible that the contribution is the result of very unequally distributed inputs from developers.

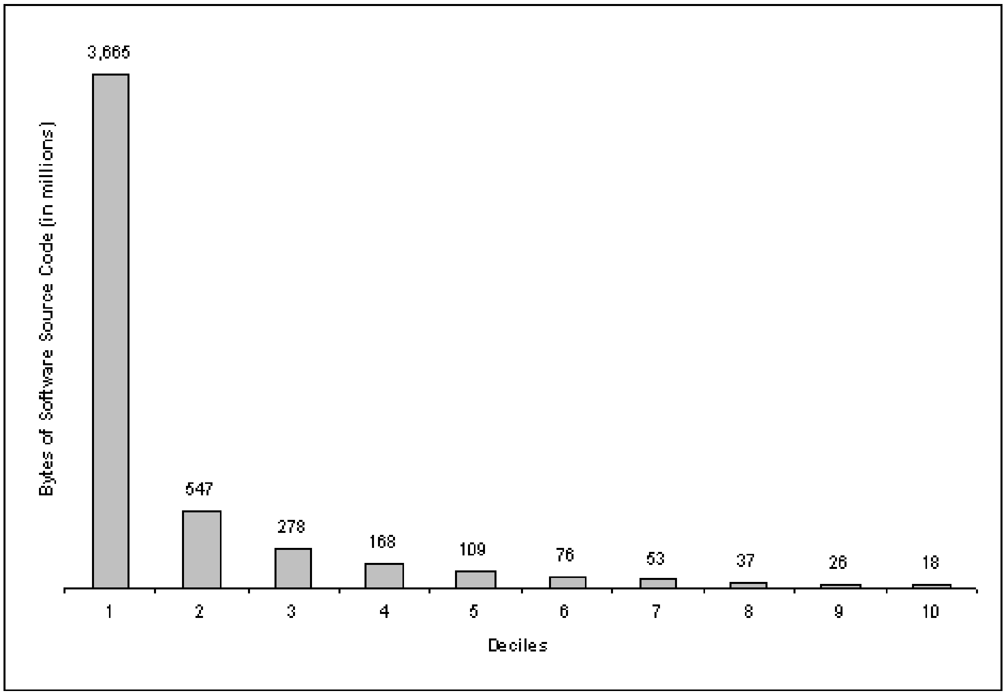
Figure 1: Authored Software Code



It is evident that a few software developers provide a large share of the software code, which is embodied by the almost vertical part of the curve on the left. Other than these very active developers, the contributions of the others decreases gradually in a smooth curve.

This result becomes clearer if we look on the distribution of the analysed software code authorship by deciles (figure 2).

Figure 2: Software Source Code Authorship by Decile



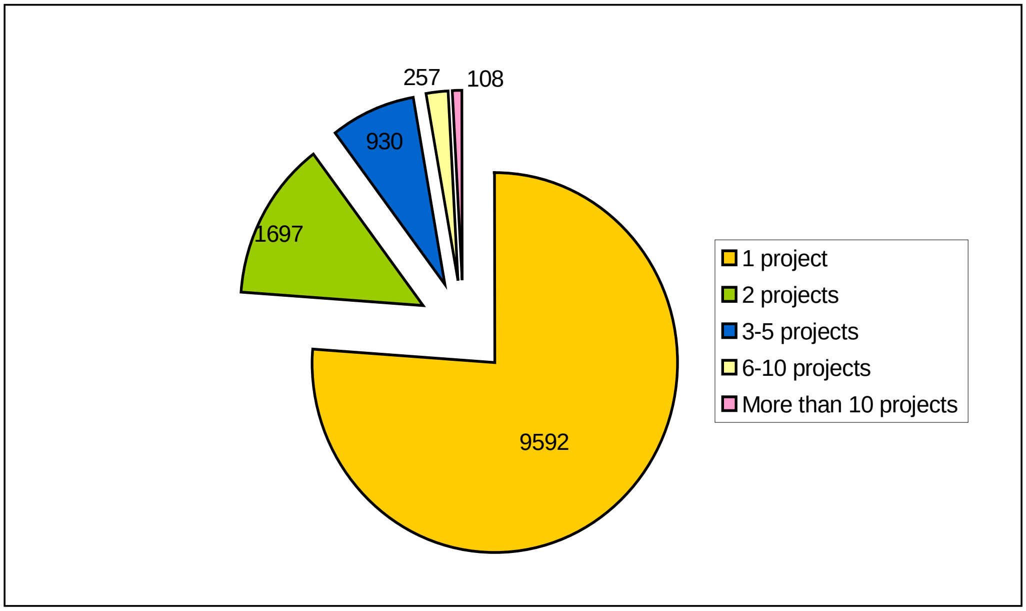
The input provided by the first decile (i.e. the top 10% developers) makes up almost three quarters (74%) of the whole amount of software code that is scrutinized here. The second deciles provides another 11%, and the third deciles adds again roughly 3% of the whole software source code. Thus, deciles 4 to 10 provide each less than 1% of the whole sum of software source code.

A small part of this can be explained by the fact that some of the "authors" at the very top (i.e. in the top 10 or so) are not actually individual authors but organizations such as the Free Software Foundation, as in many cases source code copyright is held by such an organization with absolutely no individual authorship information available. However, this does not explain most of the result, which must be simply due to the fact that the organization of collaborative OS/FS development really is quite top-heavy.

For more insight into the structure of contributions to OS/FS projects by individual authors, we utilized two thresholds to differentiate contributions further. The first was set at a level of at least 20%, the second at a level of at least 40%, both figures representing amounts contributed towards any given project. Only 39% of the author sample passed the 20%-threshold in at least one complete software project, and only 17% passed at least once the 40%-threshold.

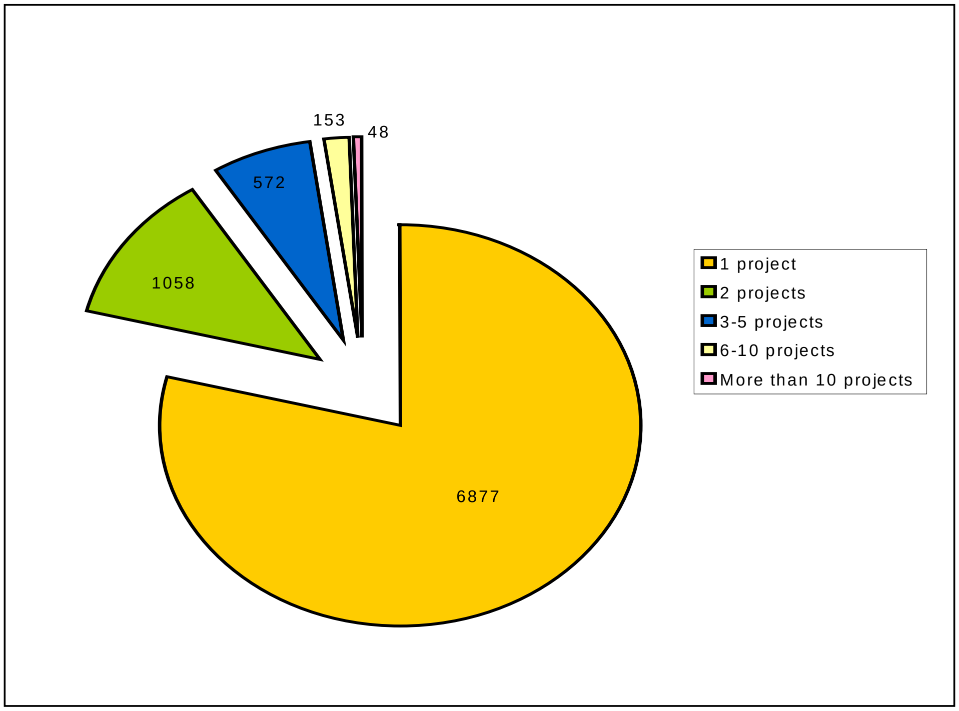
9592 (76%) of the 12584 authors who passed the 20%-limit passed this threshold only in one project, 1697 (14%) contributed at least 20% of the software code to two projects, 930 (7%) to three to five projects, and 257 (2%) reached this threshold in six to ten projects (figure 3). Only 108 OS/FS developers (1%) contributed at least 20% of the software source code to more than ten projects.

Figure 3: Number of Authors contributing at least 20% to projects



Considering only those developers who contributed at least 40% of the software source code of at least one OS/FS project, we find following distribution: 6877 (79%) contributed this share to only one project, 1058 (12%) to two projects, 572 (6.5%) to three to five projects, 153 (1.8%) to six to ten projects, and 48 (0.7%) to more than ten projects (figure 4).

Figure 4: Number of Authors contributing at least 20% to projects



This distribution of authorship contribution indicates that most authors contribute large proportions of source code to their own individual projects, or projects with the collaboration of a small number of other people, and in addition to that contribute relatively small amounts of code to larger projects.

This finding seems to be supported by an analysis of the distribution of projects among authors, as shown in the next section.

1.5. Project Size

1.5.1.OS/FS Project Structure by Size and Number of Authors

Beyond authors-based analysis, we look at data from the perspective of the projects. This examination comprises 16905 OS/FS projects, ranging from a minimum size of 69 to a maximum of 97379040 bytes of software source code. The average size (mean) of these projects is 346403.2 bytes, and on average 5.1 authors contribute to a project. However, if we consider the median values, which indicate the point where the distribution is divided into two equally large parts of the sample, we find that the distribution is, again, very one-sided. The median value for the average size of the project is only 53430 bytes, i.e. only one sixth of the mean value. The median value for the number of authors contributing to a project is 2. A large majority of OS/FS projects are small, far below the mean of 346403.2 bytes and 5.1 authors.

Figure 5 and Figure 6 illustrate this structure of OS/FS projects. 17% of the projects are smaller than 10,000 bytes, 13% lie within a range of 10,000 and 20,000 bytes, and another 19% reach a size of 20,000 to 50,000 bytes (Figure 5). Thus, almost half of the projects do not reach 50.000 bytes of software source code and remain far below the mean size. 14% of the projects have a size of 50,000 to 100,000 bytes, another 14% have a size of 100,000 to 200,000 bytes, and almost 13% of the projects are between 200,000 and 500,000 bytes. Only 13% of the projects are larger than 500,000 bytes, and it is worth noting that only 1% of the projects are very large, i.e. above 5,000,000 bytes.

Figure 5: OS/FS Projects by Size

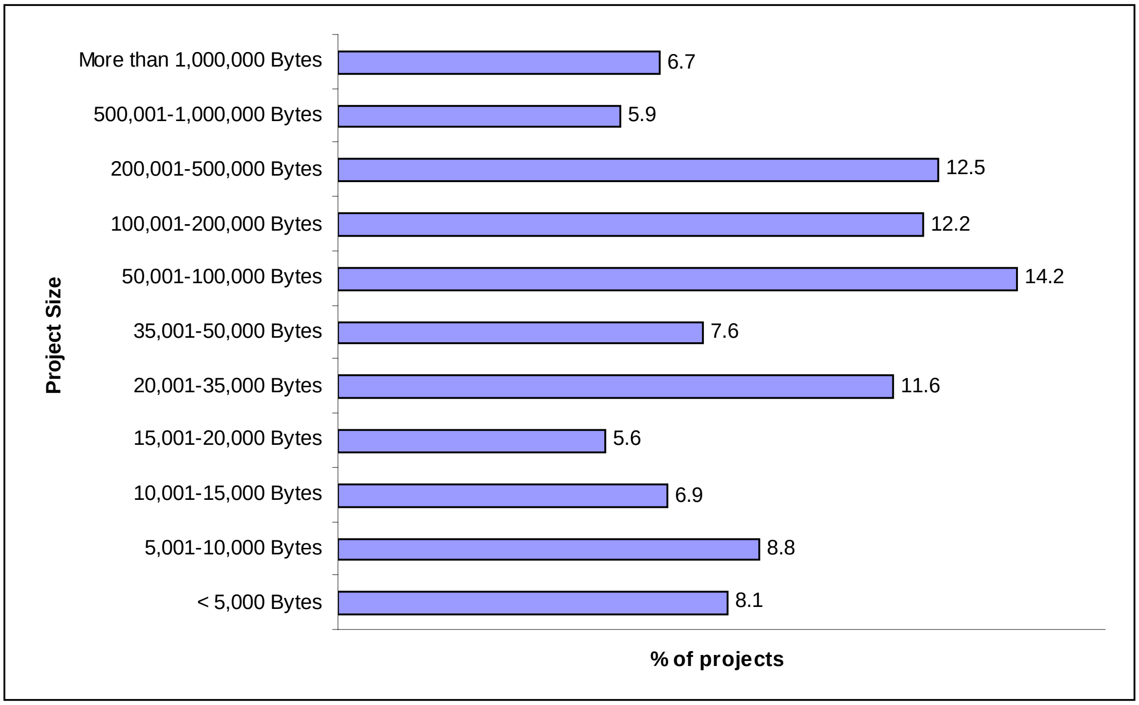


Figure 6 shows that the majority of OS/FS projects is worked on by only one or two software developers. Still, a considerable number of projects consist of three to six authors. Then, the number of authors per project decreases gradually, and we hardly find any projects at all that are performed by more than 20 software developers.[12]

[12] "0" authors in figure 2 does not mean that there were no authors at all, but that in these cases the number of authors could not be specified by the CODD analysis.

Figure 6: OS/FS Projects by Number of Authors

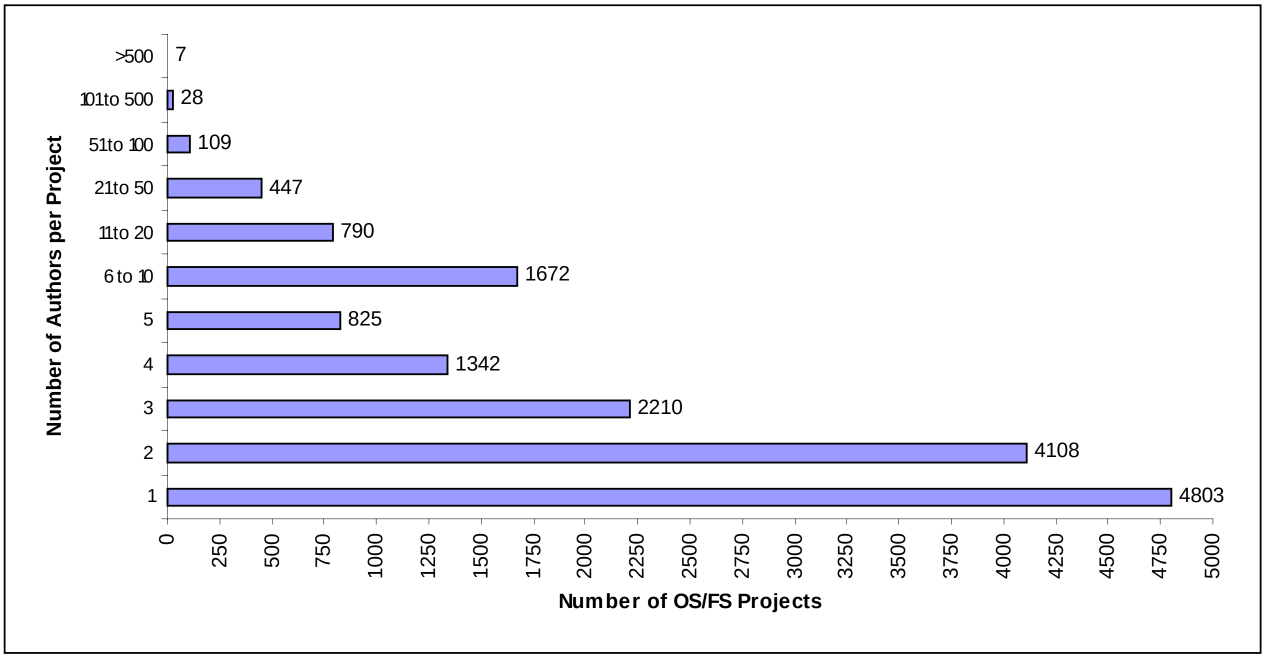
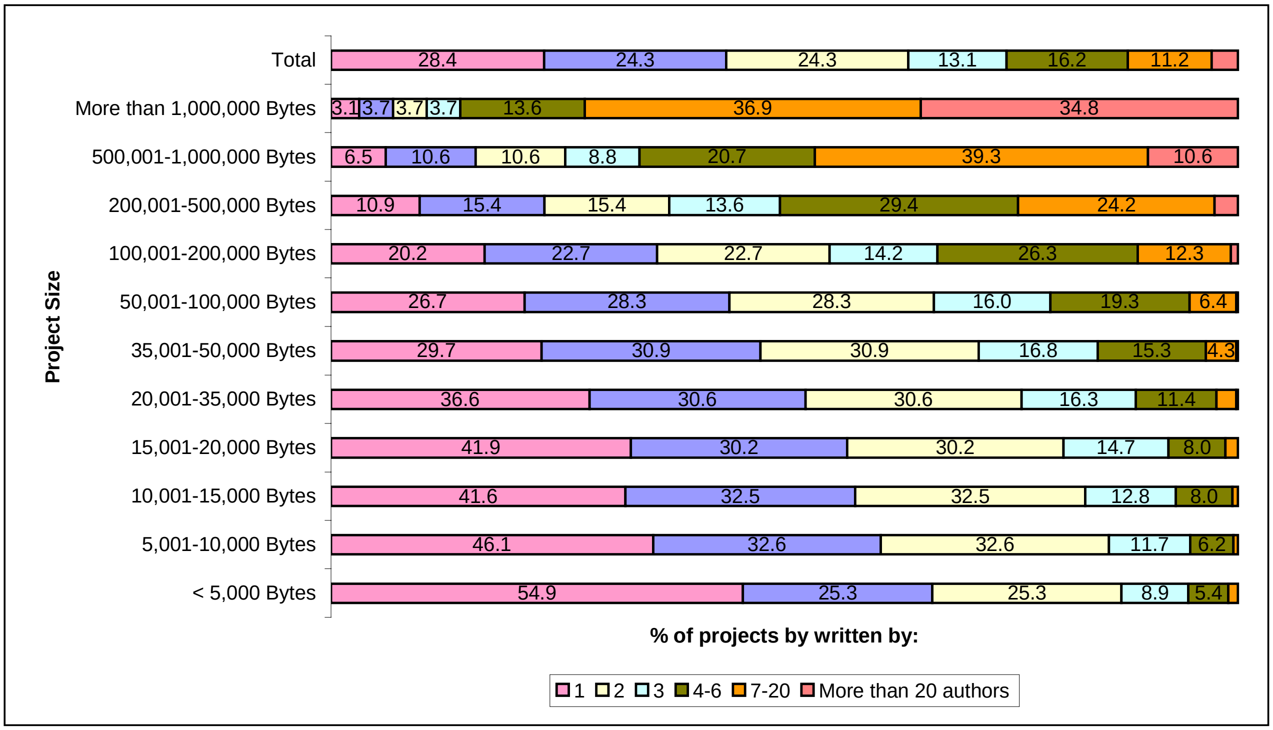


Figure 7 illustrates the relation between project size and the number of authors that contribute to it. Not surprisingly, as a general tendency we find the number of authors contributing to a project increasing with the size of the project. The scope of projects that are predominantly performed by only one author ranges from a size of 69 up to 20,000 bytes of software source code. Within this scope, we also find extraordinarily high shares of projects that are performed by two developers, but here the core ranges from projects of 5,000 to 50,000 bytes of software source code. Between projects of 20,000 and 200,000 bytes of software source code we find several projects that are performed by three developers, while projects performed by four authors are common within a range from 50,000 to one million bytes of software source code. Seven to twenty authors start collaborating on projects as small as 200,000 bytes of software source code, while more than 20 developers are typically found in projects from a size of 500,000 bytes of software source code onwards.

Figure 7: Number of contributing authors by project size

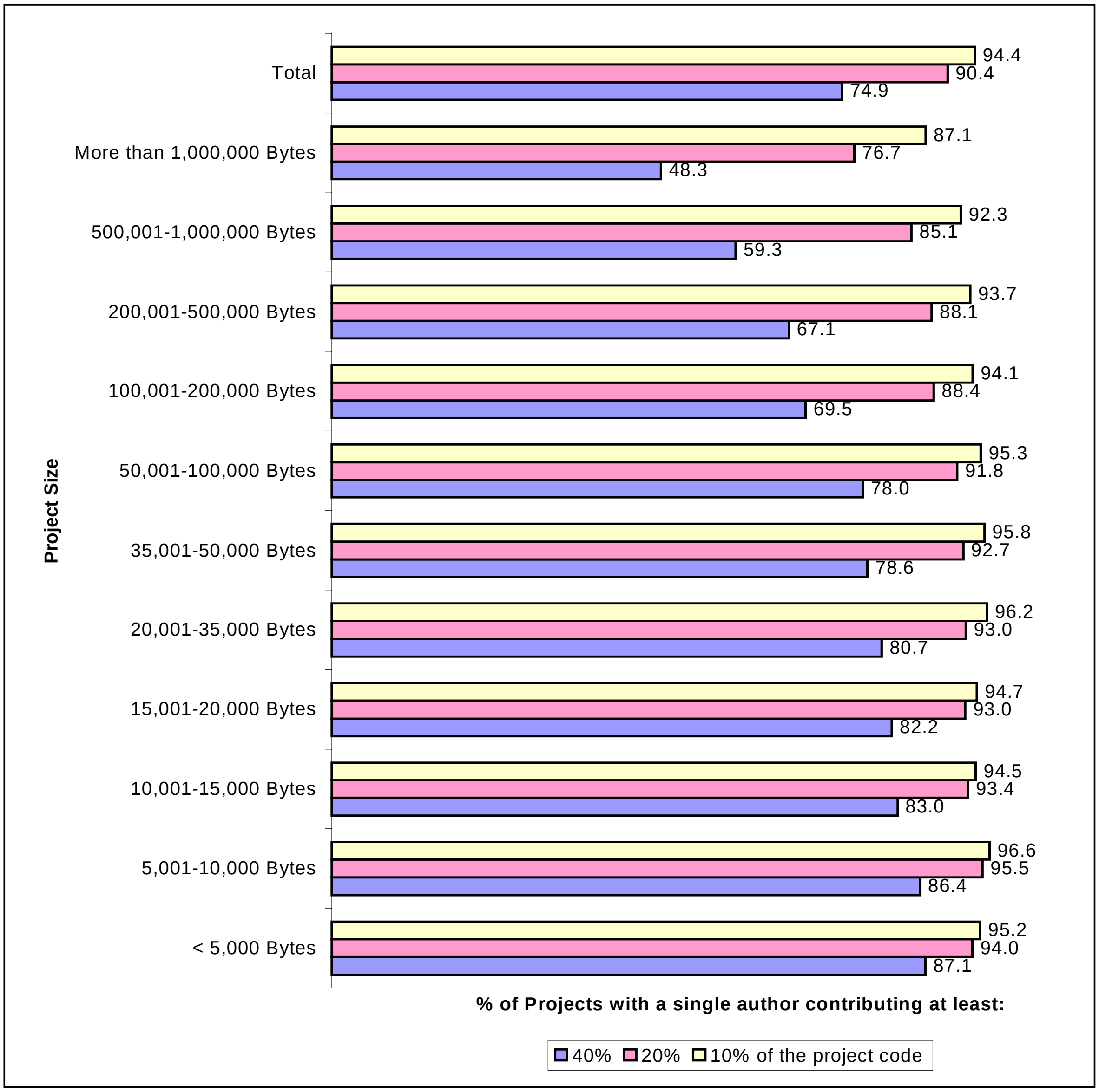


1.5.2.Structure of Authors' Contributions to OS/FS Projects

This structure is also reflected in the fact that in 94.4% of all projects we found a contribution of at least 10% of the project source code provided by a single author, in 90.4% of all projects we find at least 20% of the source code provided by a single author, and in three quarters of all projects we find a contribution of 40% of the whole software source code from one author alone (cf. the totals in Figure 8). The expected tendency of an increase in the number of authors along with increasing project size is also reflected I in Figure 8 by the decreasing shares of 40% single-author contributions to a project as the project size increases. However, even in the category of the largest projects we find that in almost half of them there is an author who has contributed at least 40% of the whole software code.

This supports the conclusion that projects are often originated by a single author and that author's contribution remains crucial even as the project grows, attracting several more contributors.

Figure 8: Contributions of single authors to projects by project size


Project Size
Total
More than 1,000,000 Bytes
500,001-1,000,000 Bytes
200,001-500,000 Bytes
100,001-200,000 Bytes
50,001-100,000 Bytes
35,001-50,000 Bytes
20,001-35,000 Bytes
15,001-20,000 Bytes
10,001-15,000 Bytes
5,001-10,000 Bytes
< 5,000 Bytes
% of Projects with a single author contributing at least:
40% 20% 10% of the project code