CORE
Metadata, citation and similar papers at core.ac.uk
Provided by University of Lincoln Institutional Repository

Structural Complexity and Decay in FLOSS Systems: An Inter-Repository Study

Andrea Capiluppi Karl Beecher
Centre of Research on Open Source Software – CROSS
Department of Computing
University of Lincoln, UK
{acapiluppi,kbeecher}@lincoln.ac.uk

Abstract

Past software engineering literature has firmly established that software architectures and the associated code decay over time. Architectural decay is, potentially, a major issue in Free/Libre/Open Source Software (FLOSS) projects, since developers sporadically joining FLOSS projects do not always have a clear understanding of the underlying architecture, and may break the overall conceptual structure by several small changes to the code base.

This paper investigates whether the structure of a FLOSS system and its decay can also be influenced by the repository in which it is retained: specifically, two FLOSS repositories are studied to understand whether the complexity of the software structure in the sampled projects is comparable, or one repository hosts more complex systems than the other. It is also studied whether the effort to counteract this complexity is dependent on the repository, and the governance it gives to the hosted projects.

The results of the paper are two-fold: on one side, it is shown that the repository hosting larger and more active projects presents more complex structures. On the other side, these larger and more complex systems benefit from more anti-regressive work to reduce this complexity.

1 Introduction

The "success" of FLOSS projects in past literature has been empirically evaluated via process and resource attributes, such as the frequency of releases, the number of developers, or by using proxies of their pool of users. In other words, the *evolvability* of FLOSS systems has been used as a term of reference for their success [10, 34, 17]. In terms of Lehman's laws of software evolution, FLOSS (E-type) systems, when growing in size and capabilities, will also grow in complexity as they evolve (according to the 6th law). They will also require increasing levels of work done to control complexity (*i.e.* "anti-regressive work", as termed in the 2nd law) and quality (as stated by the 7th law) [26] in order to maintain their evolvability. Among FLOSS well-known projects, such as the so-called LAMP stack (Linux, Apache, MySQL, Perl), the Debian family, and *BSDs, have achieved higher evolvability than others [30]. Their evolvability has been made possible by attracting a large community of users, as well as a strong and dynamic base of developers [34]. The user community initiates the need for change while the developers make it happen [33], and both are key factors in the evolution process [10].

A by-product of the continuous evolution of software systems has been identified in the deterioration of the integrity of the software. Such deterioration may manifest itself as growing complexity [25], and/or the loss of architectural and conceptual integrity [32]. This phenomenon, sometimes called *code decay* in the literature (e.g. [16]) is likely to make it progressively more difficult to understand the inner workings of the software and, hence, to implement functional additions and changes. Similarly to traditional in-house software, it has been noted that increasing the size of FLOSS systems has an effect on their underlying complexity: a growing pattern has been detected when evaluating complexity with either the traditional cyclomatic measures [27] (as studied in [8] and [22]), or counting the compilation warnings [28] as a proxy for FLOSS system's complexity. It was also observed that the anti-regressive work and size had similar growth patterns [8].

The evolvability of FLOSS systems has been found unevenly distributed when mining large repositories, such as SourceForge [17]: the vast majority of hosted projects is small, with few developers and does not go beyond the first public release [23]. On the other hand, it has been found that the repository in which they are retained acts as an exogenous factor for the evolvability of FLOSS projects [4, 7]. Results have in fact shown that, on average, different types

of repositories benefit from different levels of evolvability and success: repositories with low barriers to entry for developers (such as SourceForge) are more likely to show lower evolvability, while forges with a stricter control of access (such as the Debian distribution) have been proven to host more evolvable projects.

The present paper builds upon the concept of the FLOSS repository being a differentiating factor for project evolvability: on top of that, it argues that these exogenous factors affect other internal product attributes also. Specifically, we study two well knwon FLOSS repositories, Debian and SourceForge: it will be studied whether those repositories that experience higher levels of evolutionary activity (Debian, as found in [4]) also contain projects that have higher levels of structural complexity. We also investigate whether the effort for controlling complexity shows also differences with repositories with less activity (e.g., SourceForge).

This paper is structured as follows: section 2 introduces the definitions of the empirical study, section 3 defines the research hypotheses, the statistical tests and the metrics used to reject or not the null hypotheses. Section 4 will present the results of these tests, and the description of what differences were detected when comparing the structural characteristics of different FLOSS repositories. Section (threats) will encompass the threats to validity of the empirical study, while section 6 concludes, presenting also potential future works.

2 Empirical Study Definition and Planning

This section introduces the definitions used in the following empirical study and presents the general objective of this work, and it does that in the formal way proposed by the *Goal-Question-Metric* (GQM) framework [3]. The GQM approach evaluates whether a goal has been reached, by associating that goal with questions that explain it from an operational point of view, and providing the basis for applying metrics to answer these questions. This study follows this approach by developing, from the wider goal of this research, the necessary questions to address the goal and then determining the metrics necessary for answering the questions. By following the GQM, two research questions will be proposed, that in turn will be distilled into hypotheses to be statistically tested in the next section 3.

Goal: The long term objective of this research is to understand whether different FLOSS repositories receive different levels of evolutionary activity, and whether this fact is correlated to different levels of complexity. In terms of Lehman's laws, the goal is to demonstrate that different FLOSS repositories face different levels of complexity.

Question: Two research questions are proposed:

- Do the projects in each repository have significantly different levels of complexity on average?
- And, if there is such difference, is there a correlation between a larger evolvability of FLOSS projects and the amount of work done to control their complexity (*i.e.*, anti-regressive work)?

Metric: Source code is organized in *files* which provide both creation and storage units and separate the total code into portions that can be separately compiled. In general, within these files, programmers create functions, procedures or methods (depending on the programming language) that perform operations on the data, so that the desired computations are performed. The various functions contained in the same file may differ in dimension and other attributes, and provide a good trade-off between too fine (as in "lines of code") and too course (as in "source files") granularity. Based on this trade-off, functions are selected as the basic unit of study in our research. We take advantage that there are simple and well-defined approaches to measure complexity at the function level. All the metrics used in this study will be focused on the level of granularity of source functions (or methods).

2.1 Definitions

The attributes considered in the present study are described below:

Coupling: this attribute measures the degree to which each source element relies on other elements (i.e. how interconnected is the code) [13, 1]. Since this study is conducted at the function level, the union of all the function calls (and method invocations) form the network of couplings in a system. Each coupling can be uniquely categorized as inbound (c_in) or outbound (c_out) (or fan-in and fan-out), depending on the direction of the call. For example, function "sign off" (Figure 1) has two inbound (fan-in) and three outbound (fan-out) couplings.

Instability: from past literature, instability is the ratio of fan-out coupling (c_o) to total coupling (c_o + c_i) such that $I = \frac{c_o}{(c_o + c_i)}$. This metric is an indicator of the resilience to change of software components (as in source functions) [21]. The range for this metric is 0 to 1, with $I = 0$ indicating the lowest instability for an element and $I = 1$ indicating a completely unstable element. The example in figure 1 therefore has $I = 0.6$.

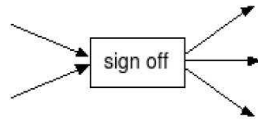

Figure 1. example of 'inbound' and 'outbound' couplings for the function "sign off"

2.2 Motivation

This work is based upon previous work carried out by the authors [5]. In that work, six FLOSS repositories were studied to test the hypotheses that they could be considered to belong to a type based on their characteristics, and that each type attracted different rates of development. As a result, FLOSS repositories were identified as exogenous drivers that influence the evolutionary activity of their member projects. The test was done by taking a sample of 50 projects from each repository (totalling 300 projects in all) and comparing four historical indicators for each project: commit rates, number of commiters, size, and age.

From the repositories examined, three repository types were proposed and defined (open host, metaproject, and distribution). It was also pointed out (after a related study [4]) that a project may transition between repositories (illustrated in figure 2), and that its resulting evolutionary activity can change in a way consistent with its new host repository.

While the previous works examined process metrics, this work is the first step to adapting the same investigative approach to product metrics. As stated in the goal above, this line of work will ascertain if the rates of evolutionary activity already identified are related to the resulting product complexity. This first step will examine just two repositories (Debian and SourceForge), and will be expanded to a greater number of repositories in later work.

2.3 Repository Samples

The two FLOSS repositories to be mined (Debian and SourceForge) are introduced here. A sample of 50 individual projects was chosen from each repository by a randomizer, and each project was checked to ensure that it was not also available in the counterpart repository. A checkout was then performed on each member project of each sample's development trunk (from either their CVS or SVN source control repositories); any branches were ignored. The list of analyzed projects is shown in Table 6. Each of these projects was then analyzed to obtain the metrics needed to perform the investigation; the measures for the study are those introduced above, and summarized in the columns of Table 6.

Both Debian and SourceForge allow administrators to label their project's development status. In order to compare projects with a similar status, only the "stable" branch of the repositories was considered for study. Considering projects with similar status means highlighting the most evolvable projects in both samples; but also dealing with pools of similar sizes, thus avoiding inaccurate comparisons.

Debian A distribution of Linux that hosts a large number of FLOSS projects. At the time of writing more than 20,000 projects are listed under the "stable" label.

SourceForge.net A well-known repository enabling the hosting and management of a wide variety of free software projects. At the time of writing there are over 20,000 projects in the with the label "production/stable" alone.

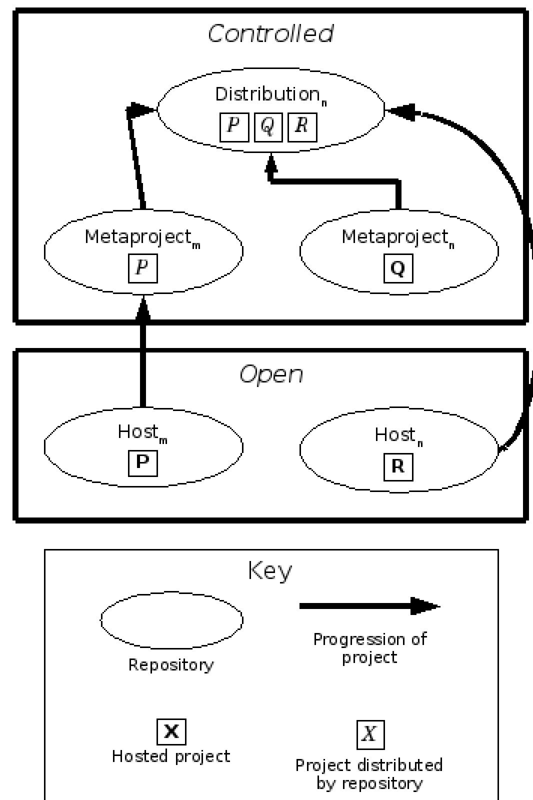

Figure 2. FLOSS forges and progression of projects between them

3 Research Hypotheses

In this section, two research hypotheses are proposed: each is then evaluated using a statistical significance test,

allowing either a retention of the null hypothesis (H_0), or a rejection of it in favour of the alternative hypothesis (H_1). The hypotheses, metrics and statistical tests used are summarized in Table 1.

3.1 Hypothesis 1

The first hypothesis concerns the structural complexity of the interconnected components that form each of the project's source code. It postulates that each repository will contain projects that, on average, possess a level of structural complexity consistent with the amount of evolutionary activity it receives. The provisions for testing this hypothesis are shown in table 1.

		Hypothesis	*Test*
Hypothesis 1	H_0	Debian and SourceForge projects have significantly different levels of structural complexity.	$I_d = I_s \cap$ $Fin_d = Fin_s \cap$ $Fout_d = Fout_s$
	H_1	There is no significant difference between the levels of structural complexity of Debian and SourceForge projects.	$I_d \neq I_s \cap$ $Fin_d \neq Fin_s \cap$ $Fout_d \neq Fout_s$
Hypothesis 2	H_0	Debian projects receive significantly different rates of control work to SourceForge projects.	$CW_d = CW_s$
	H_1	Debian and SourceForge projects receive similar rates of control work.	$CW_d \neq CW_s$
Keys			
		$I = Instability$	
		$Fin = Fan\ in$	
		$Fout = Fan\ out$	
		$CW = Control\ Work$	

Table 1. Summary of the hypotheses to be tested

For each project, all functions are extracted and their fan-in, fan-out and instability calculated, and the three summary statistics also calculated (mean, maximum and variance). Each summary statistic is then combined, per repository, into a distribution to allow comparison of samples.

3.2 Hypothesis 2

The second hypothesis concerns the work done to organize and control the structural complexity of the components (i.e., source functions). It postulates that each repository will contain projects that, on average, have received an historical level of work done to its structure that is consistent to the amount of evolutionary activity it receives. That is to say, repositories that have a history of greater evolutionary activity contain projects that have received more work done to control their structural complexity than repositories receiving lesser evolutionary activity.

To test this hypothesis, projects from both repositories will each have a series of uniformly spaced snapshots taken from their history: this will be achieved by dividing the number of commits in three distinct periods. Given the number of commits N, the three snapshots are taken in these dates:

FP: First point; the snapshot taken on the date of commit number $\frac{N}{3}$[1].

MP: Mid point; the snapshot taken on the date of commit number $\frac{2N}{3}$.

LP: Last point; the latest available snapshot (i.e., on the date of commit number N).

The overall number of commits for all the projects is reported in Table 6.

Within each snapshot, each function's fan-out and instability was compared to its counterpart in the succeeding snapshot, where a counterpart exists. Functions that have reduced in value between snapshots are counted: the control (*i.e.*, anti-regressive) work between two snapshots is the rate of reduction in these values. Preferably, the number of intervals should be larger to provide more precise results, and it is intended that further work will address this. However, three points provides a simple minimum that determines if further, more fine-grained investigation is warranted.

4 Results

Hypothesis 1 This section summarizes the findings that were collected by evaluating the first research hypothesis. The attributes presented above were evaluated at the latest available change recorded in the CVS or SVN repositories. For each project in the two samples, the mean and variance values of all the measures were used to compare the forges. The maximum value was also used where appropriate.

To determine the statistical test to be used, a Kolmogorov-Smirnov test was applied to the data sets to establish a goodness of fit to the normal distribution, using the R programming language [14]. The distributions were found to be non-parametric, so the *Wilcoxon unpaired rank-sum test* (also known as the Mann-Whitney U test), a statistical significance test for non-parametric data, was selected.

Table 2 shows the results of the significance tests; these are one-tailed tests assuming that Debian possesses larger

[1] This arrangement was chosen instead of selecting snapshots 1, $\frac{N}{2}$, and N because some projects in the sample begin with either little or no material to analyze within their source repositories.

	Mean	Maximum	Variance
Fan-in	$\mathbf{p = 0.0003}$	$\mathbf{p = 0.0012}$	$\mathbf{p = 0.019}$
Fan-out	$\mathbf{p = 0.0003}$	$\mathbf{p < 0.0001}$	$\mathbf{p < 0.0001}$
Instability	$p = 0.09$*	-	$\mathbf{p < 0.0001}$*

Table 2. Results of p-values for hypothesis 1, one-tailed test

values, excepting * values, where the reverse is true. They show that Debian projects can be considered more complex in terms of functional coupling due to their greater interconnectedness. However, the extent to which they are significantly more complex is questionable. Whilst Debian projects consistently display levels of coupling that are more excessive and within a much greater range, the mean values do not differ consistently.

Hypothesis 2 This hypothesis builds upon the evolution of the projects within the repositories, and three snapshots in time (as detailed above in section 3.2) of the source code are taken and analyzed. It was hoped to apply this analysis to all projects under investiation; however, the amount of data storage and processing required per project was very large. Given this eventuality, coupled with the authors' planning to extend this investigation to other repositories, it was decided to take only the five largest projects from each sample and split each project into three uniformly spaced points from an evolutionary point of view. Table 3 summarizes the evolution (in terms of SLOCs) of the five largest projects within SourceForge and Debian in the three selected points in time. Major changes can be noted in the Debian sub-sample, up to a maximum of 500% of the initial size (e.g., the boson project); whereas the projects within the SourceForge set have less evolvability, with a maximum increase of 30% of the initial size (the moses project).

		FP	MP	LP
sf.net	fsdb	238,673	228,739	241,218
	moses	81,289	99,791	105,955
	ozone	73,755	89,061	63,790
	QPolymer	87,248	86,939	86,971
	xqilla	91,884	92,682	107,320
debian	boson	37,325	124,116	224,567
	dia	95,191	120,736	146,550
	openafs	522,667	595,220	618,553
	openh323	52,014	84,045	234,285
	Pike	73,377	143,906	173,196

Table 3. Growth in size – SLOCs

As discussed above, the investigation of the amount of anti-regressive effort by the FLOSS developers into their systems involved the comparison of the two attributes fan-out and instability in three temporal points (FP, MP, LP). Specifically, the reduction of these two attributes between subsequent points was recorded twice (FP → MP, and MP → LP, respectively), and the results reported in Table 4.

As visible in the table, the sub-sample of Debian projects consistently receive a greater amount of architectural control work: on average, the fan-out was more reduced in the Debian subsample than in SourceForge subsample in the comparison both FP → MP (5% against 2.2%) and MP → LP (6.5% against 2.24%). Also the instabilities reflect this divide, both in the first comparison (on average, the 8.9% of the Debian subsample received a reduction of instability, against the 1.51% of SourceForge), and in the second one (7.36% against 1.23%). Interestingly, the one project in the SourceForge sample that contradicts this tendency has recently, since the data was gathered, been introduced into the **unstable** branch of Debian, rather than the stable branch under investigation.

Further to the total control work performed, the effort given to curbing *excessive* fan-out was also investigated. A fan-out value of greater than 7 was designated as excessive [29]; similarly to the work shown in table 4, the proportion of excessive functions brought below this threshold between the historical snapshots in each project was calculated (see table 5). As a general result, the same above trends appear here as well: debian achieves an average of 9.52% in FP → MP and 10.90% in MP → LP and less variance; SourceForge achieves smaller amounts, on average 7.88% in FP → MP and 6.52% in MP → LP, but a larger variance.

		FP → MP		MP → LP	
sf.net	fsdb	1/407	0.2%	5/375	1.3%
	moses	1/81	1.2%	1/115	0.90%
	ozone	10/93	10.8%	6/81	7.4%
	QPolymer	2/14	14.3%	0/12	0.0%
	xqilla	17/132	12.9%	19/83	23.0%
debian	boson	6/39	15.4%	67/215	31.2%
	dia	3/118	2.5%	8/183	4.4%
	openafs	48/781	6.1%	22/822	2.7%
	openh323	5/43	11.6%	6/87	6.9%
	Pike	7/59	12.0%	10/107	9.3%

Table 5. Number of functions with excessive fan-out reduced

5 Related Work

This paper relates mainly to other works that have addressed FLOSS from an evolutionary perspective and have attempted to evaluate it using software metrics.

		Fan-out				Instability			
		FP → MP		**MP → LP**		**FP → MP**		**MP → LP**	
Debian	boson	41/1000	4.1%	731/4839	15.1%	73/1000	7.3%	620/4839	12.8%
	dia	81/2864	2.8%	62/3642	1.7%	204/2864	7.1%	199/3642	5.5%
	openafs	698/10020	7.0%	293/10555	2.8%	892/10020	8.9%	423/10555	4.7%
	openh323	47/1281	3.7%	67/2168	3.1%	81/1281	6.3%	72/2168	3.3%
	Pike	107/1450	7.4%	215/2201	9.8%	216/1450	14.9%	231/2201	10.5%
	Averages		**5.0%**		**6.5%**		**8.9%**		**7.4%**
SourceForge	fsdb	30/8836	0.30%	107/8755	1.2%	43/8836	0.50%	57/8755	0.65%
	moses	28/3278	0.85%	15/3863	0.34%	49/3278	1.5%	33/3863	0.85%
	ozone	111/5027	2.2%	82/5903	1.4%	109/5027	2.2%	76/5903	1.3%
	QPolymer	4/648	0.62%	3/650	0.46%	5/648	0.77%	5/650	0.77%
	xqilla	132/1896	7.0%	145/1847	7.8%	50/1896	2.6%	48/1847	2.6%
	Averages		**2.2%**		**2.2%**		**1.5%**		**1.2%**

Table 4. Work done to reduce coupling between snapshots (number of functions with fan-out and/or instability reduced)

The evolution of software is a research area with a strong history in software engineering, and research into FLOSS-specific evolution has been an area of growing significance since works such as Godfrey and Tu's [20] demonstration of the growth of the Linux kernel that ran counter to traditional software engineering experience. Since the beginning of the 2000s a number of works have investigated the evolutionary dynamics of software projects developed in a distributed heterogeneous environment and have addressed a wide spectrum of attributes. As well as the analysis of process and resource attributes, the free availability of development artefacts has allowed the relatively easy examination of how product attributes change over time, including the product size [15, 18, 20, 24], complexity [9], and the nature of its structure [9, 19].

This paper specifically addresses the structural characteristics of FLOSS, explicitly the organization of the software's constituent components. A number of measures for doing this exist and, in past work, such characteristics have been examined at varying levels of granularity and with consideration to specific paradigms. After the development of structured programming relatively simple measures of modular structure were developed and refined. For example, coupling (a simple count of connections that measures the interconnectedness of a module [12]) was refined into a more complicated form, whereby a connection was judged qualitatively [31]; this allows a "strength" to be assigned to a connection, but is a somewhat informal measure that does not lend itself easily to automated measurement. Similar developments have occurred with respect to measures that are specific to object-oriented design only [11, 6]. Structural metrics, like these examples, have been used as measures to investigate the affects of interconnectedness on maintainability attributes such as ripple effects [35], fault-proneness [2] and maintenance effort [11].

Many existent FLOSS projects belong to some type of repository, i.e. an organized central location for the purpose of managing and controlling software development. Most previous empirical works in FLOSS have used a small number of exemplary projects, thereby affecting the generalizability of their findings, but some works have taken large samples of projects [15, 17]. These samples have normally been taken from a specific repository, whereas this work builds on previous works by the author [5, 7] in which samples are taken from multiple distributions in order to perform a comparison of them using statistical significance testing. Part of this line of research shows that repositories can be considered as being one of a set of types, and each type tends to undergo a particular level of evolutionary activity as evidenced by the average size, age, number of developers and rate of commits. Projects may originate in any repository type and transition between them.

6 Conclusions and Future Works

This paper has been built on top of existing results [4, 5], claiming that the rate of evolvability of a FLOSS project can be influenced by the repository it resides in. By carrying out a comparative analysis between two FLOSS repositories (Debian and SourceForge), it evaluated the architectural structure of a sample of projects from each, and argued that the same structure is influenced by the repository itself, as an exogenous factor. Three attributes were used as proxies of architectural structure: fan-in, fan-out and instability.

A static comparison was performed between the two samples to ascertain if any absolute differences existed between them. Results from this investigation showed that the projects residing in Debian typically had components

with excessive and more varied levels of interconnectedness. Apart from a low significance in the instability (i.e., 90%), it could also be satisfactorily concluded that the average values differed significantly.

A second investigation was also carried out: a sub-sample of five of the largest projects was taken from each sample and its characteristics studied in three subsequent temporal points (FP, MP, LP), equally spaced; specifically, the work done to reduce the two attributes "fan-out" and "instability" was measured between FP and MP, and between MP and LP. Results showed that the proportion of functions that received complexity control work was consistently greater in the case of Debian (apart from a single SourceForgeproject that, interestingly, has now been introduced into the unstable branch of the Debian project).

The two results as summarised here confirm the earlier working hypothesis: FLOSS repositories act as exogenous factors in the evolution of the projects they contain. In fact, the greater effort and evolvability observed in Debian projects [4] is mirrored by the amount of complexity control done at the architectural level, when comparing it with SourceForge.

As future works, two main areas have been considered: at first, this work will be extended by including more repositories in the investigation. Specifically, more representatives for each of the repository types identified in figure 2 will be will not only ascertain the architectural evolution of metaprojects, but could also lend more empirical weight to the generalizability of the findings.

In addition, the findings of the hypothesis 2 will need to be investigated further by decreasing the granularity of the intervals in order to provide a more precise set of results. A snapshot every month will be considered in order to give more visibility to phases of maintenance and refactoring, which could bias the results. When refactoring is followed by another type of maintenance, the latter might increase complexity again. The three intervals as above would be too coarse, and produce different results depending on whether the analysed point was before or after these activities.

7 Threats to Validity

- **Construct validity** The study has been able to make use only of available data. It is possible, for example, that the project initialization pre-dates the first measurable piece of historical data and is therefore beyond the reach of our analysis. Further to this point, the development status assigned to a project requires human judgement. Whilst in the Debian project this is a very systematically prescribed process, the status assigned to a SourceForge project is the result of the subjective judgement of the administrator. There is therefore the risk that some projects in the SourceForge sample are not fairly considered "stable".
- **Internal validity** The permissive nature of FLOSS development means that it is possible, even encouraged, for individual projects, or parts of them, to be included in more than one repository. Hence, when randomly sampling projects from individual repositories, it is possible that a sampled project may be found in another unknown location and that its evolution is also influenced by this unknown repository. The assumption is therefore made that any such confounding effect, if present, is negligible.
- **Internal validity** Because of the large amount of analysis necessary the evaluation of coupling was automated by analysis software, which is presently at a level of sophistication that has the following consequences: 1) Test suites, when included within the software package, are included in the analysis and so contribute to the perceived level of structuredness. It is arguable whether or not test suites should be considered "part of the software". For example, in the Debian sample, 14 out of 50 projects had a directory that appeared to function as a test suite; 2) The level of coupling is limited to being derived from a static view of the software; hence, dynamic coupling is not detected.
- **External validity** Further to the previous point, the large amounts of analysis and storage space required meant that the investigation into hypothesis 2 required a sub-sample to be taken rather than both samples in its entirety. It is desireable for future work to explore ways in which analysis can be made much more efficient and so allow the analysis of a greater number of projects.

References

[1] E. A. Audun Foyen and L. C. Briand. Dynamic coupling measurement for object-oriented software. *IEEE Trans. Softw. Eng.*, 30(8):491–506, 2004.

[2] V. R. Basili, L. C. Briand, and W. L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, pages 751–761, October 1996.

[3] V. R. Basili, G. Caldiera, and D. H. Rombach. The goal question metric approach. In *Encyclopedia of Software Engineering*, pages 528–532. John Wiley & Sons, 1994. See also http://sdqweb.ipd.uka.de/wiki/GQM.

[4] K. Beecher, C. Boldyreff, A. Capiluppi, and S. Rank. Evolutionary success of open source software: An investigation into exogenous drivers. *Electronic Communications of the EASST*, 8, 2008.

[5] K. Beecher, A. Capiluppi, and C. Boldyreff. Identifying exogenous drivers and evolutionary stages in floss projects. *Journal of Systems and Software*, 2009.

[6] L. C. Briand, J. W. Daly, and J. Wust. A united framework for coupling measurement in object-oriented systems. *IEEE Transactions on Software Engineering*, pages 91–121, January/February 1999.

[7] A. Capiluppi, C. Boldyreff, and K. Beecher. Quality factor and coding standards – a comparison between open source forges. In *Proceedings of the Second International Workshop on Software Quality and Maintainability*, 2008. Colocated with the 12th European Conference of Software Maintenance and Reengineering.

[8] A. Capiluppi and J. Fernández-Ramil. Studying the evolution of open source systems at different levels of granularity: Two case studies. In *IWPSE '04: Proceedings of the Principles of Software Evolution, 7th International Workshop*, pages 113–118, Washington, DC, USA, 2004. IEEE Computer Society.

[9] A. Capiluppi and J. Fernández-Ramil. A model to predict anti-regressive effort in open source software. In *23rd IEEE International Conference on Software Maintenance*, pages 194–203. IEEE, 2007.

[10] A. Capiluppi and M. Michlmayr. From the cathedral to the bazaar: An empirical study of the lifecycle of volunteer community projects. In J. Feller, B. Fitzgerald, W. Scacchi, and A. Sillitti, editors, *OSS*, volume 234 of *IFIP*, pages 31–44. Springer, 2007.

[11] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, pages 476–493, June 1994.

[12] L. Constantine and E. Yourdon. *Structured Design*. Prentice-Hall, 1979.

[13] S. D. Conte, H. E. Dunsmore, and Y. E. Shen. *Software engineering metrics and models*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1986.

[14] P. Dalgaard. *Introductory Statistics with R*. Springer, 2002.

[15] A. Deshpande and D. Riehle. The total growth of open source. In B. Russo, E. Damiani, S. A. Hissam, B. Lundell, and G. Succi, editors, *OSS*, volume 275 of *IFIP*, pages 197–209. Springer, 2008.

[16] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus. Does code decay? assessing the evidence from change management data. *IEEE Trans. Softw. Eng.*, 27(1):1–12, 2001.

[17] R. English and C. Schweik. Identifying success and tragedy of floss commons: A preliminary classification of sourceforge.net projects. In *Proceedings of the 1st International Workshop on Emerging Trends in FLOSS Research and Development*, Minneapolis, MN, 2007. ICSE.

[18] D. M. German. An empirical study of fine-grained software modifications. *Empirical Software Engineering*, 11(3):369–393, 2006.

[19] D. M. German and A. Hindle. Measuring fine-grained change in software: Towards modification-aware change metrics. In *METRICS '05: Proceedings of the 11th IEEE International Software Metrics Symposium*, page 28, Washington, DC, USA, 2005. IEEE Computer Society.

[20] M. W. Godfrey and Q. Tu. Evolution in open source software: A case study. In *Proceedings of the International Conference on Software Maintenance*, pages 131–142, 2000.

[21] I. Gorton and L. Zhu. Tool support for just-in-time architecture reconstruction and evaluation: an experience report. In *Proceedings of the 27th International Conference on Software Engineering*, pages 514–523, St. Louis, Missouri, USA, 2005.

[22] I. Herraiz, J. M. Gonzalez-Barahona, and G. Robles. Towards a theoretical model for software growth. In *MSR '07: Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 21, Washington, DC, USA, 2007. IEEE Computer Society.

[23] J. Howison and K. Crowston. The perils and pitfalls of mining sourceforge. In *Proceedings of the international workshop on mining software repositories (msr 2004*, pages 7–11, 2004.

[24] S. Koch. Evolution of open source software systems – a large-scale investigation. In *Proceedings of the First Int. Conf. on Open Source Systems*, pages 148–153, July 2005.

[25] M. M. Lehman. Programs, cities, students, limits to growth? *Programming Methodology*, pages 42–62, 1978.

[26] M. M. Lehman, J. F. Ramil, P. D. Wernick, D. Perry, and W. M. Turski. Metrics and laws of software evolution - the nineties view. In *Proceedings of the 4th International Symposium on Software Metrics*, pages 20–32, 1997.

[27] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, pages 308–320, December 1976.

[28] T. Mens, J. Fernandez-Ramil, and S. Degrandsart. The evolution of Eclipse. In *Proc. International Conference on Software Maintance (ICSM)*, pages 386–395. IEEE Computer Society Press, 2008.

[29] G. A. Miller. The magical number seven, plus or minus two: Some limits on our capacity for processing information. *The Psychological Review*, 63:81–97, 1956.

[30] A. Mockus, R. T. Fielding, and J. D. Herbsleb. Two case studies of open source software development: Apache and mozilla. *ACM Transactions of Software Engineering and Methodology*, pages 309–364, July 2002.

[31] M. Page-Jones. *The Practical Guide to Structured Systems Design*. Yourdon Press, 1980.

[32] D. L. Parnas. Software aging. In *ICSE '94: Proceedings of the 16th international conference on Software engineering*, pages 279–287, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[33] G. Robles, J. J. Amor, J. M. Gonzalez-Barahona, and I. Herraiz. Evolution and growth in large libre software projects. In *Eighth International Workshop on Principles of Software Evolution*, pages 165–174, 2005.

[34] G. Robles and J. M. Gonzalez-Barahona. Contributor turnover in libre software projects. In E. Damiani, B. Fitzgerald, W. Scacchi, M. Scotto, and G. Succi, editors, *OSS*, volume 203 of *IFIP*, pages 273–286. Springer, 2006.

[35] F. G. Wilkie and B. A. Kitchenham. Coupling measures and change ripples in C++ application software. *Journal of Systems and Software*, 52:157–164, 2000.

Debian				SourceForge			
Name	Functions	SLOCs	Commits	Name	Functions	SLOCs	Commits
EtoileWildMenus	2	1,711	120	Beobachter	94	2,715	378
Pike	2,302	173,196	21,608	QPolymer	652	86,971	491
ProofGeneral	0	48,692	14,360	audiobookcutter	34	4,229	1,127
acpidump	53	2,349	36	blob	496	22,056	2377
apmud	45	2,502	74	cdlite	29	1,116	18
boson	9,246	224,567	33,216	clinkc	919	25,846	2501
cdparanoia	211	9,182	414	cotvnc	789	37,455	1,959
cherokee	1,221	54,229	5,341	cpia	109	22,954	386
clamav	1,056	116,731	11,691	critical_care	1,051	38,994	1,710
dia	4,151	146,550	22,995	csUnit	96	16,241	2,149
enigmail	86	10,790	4,387	eas3pkg	69	43,724	310
flac	1,380	56,293	9,584	edict	0	2,556	84
fte	1,182	51,498	1,941	expreval	66	3,588	139
geomview	2,748	101,844	8,637	fitnesse	2,321	39,503	5,281
gosa	2,404	107,798	21,013	fn-javabot	279	10,142	1,732
grass6	1,650	107,648	2,941	formproc	134	3,514	1,340
grub	0	3,536	1,786	fourever	593	15,163	1,970
gwenview	128	4,580	558	freemind	1,579	28,519	11,109
jToolkit	0	4,156	251	fsdb	8,506	241,218	8,325
kmouth	99	5,240	13,954	galeon	3,525	93,374	29,760
kphoneSI	735	41,829	2,152	gvision	0	101,123	1,064
libax25	80	11,721	152	hge	800	45,654	1,185
liboil	730	52,996	3,798	icsDrone	33	1,411	153
libsoup	494	15,012	2,818	intermezzo	522	34,792	2,247
lirc	785	44,753	3699	jtrac	12,771	519	4,051
myphpmoney	153	19,434	2,298	juel	404	7,284	999
netpanzer	2,935	74,368	9681	kpictorial	18,214	21	341
noteedit	611	63,456	59	modaspdotnet	45	2,445	498
octave-forge	0	78,150	21,838	moses	4,053	105,955	5,172
openafs	10,807	618,553	50,176	neocrypt	21	2,135	110
openh323	6,392	234,285	9,352	ogce	13,960	350,490	19,058
peercast	818	22,543	1912	oliver	9	1,429	187
prcs1	663	37,360	918	ozone	3,920	63,790	6,110
preludemanager	304	10,854	3,398	perpojo	117	1,677	72
radiusd	1,330	95,967	19144	pf	84,489	213	3,209
rlplot	1,449	69,493	1674	qlc	890	26,452	1575
ruby	5,086	419,942	162,369	seagull	878	54,155	1,039
scid	1,179	89,402	676	simplexml	65	1,691	66
shorewall	0	25,159	19,275	source	162	8,786	692
slrn	1,189	42,993	1843	sudapix	15,747	234	331
sylpheed	2,859	106,087	12,359	swtjasperviewer	129	3,214	204
synce-kde	141	21,684	1,392	symbolica	67	2,623	250
tcl	2,205	165,306	42,237	tab-2	597	19,067	3202
tdb	133	3,942	296	txt2xml	61	1,345	157
tiobench	41	1,689	106	ustl	684	11,416	2,223
txt2html	0	3,623	131	whiteboard	202	4,910	51
vlc	6,250	401,256	65,098	wxactivex	37	3,264	61
wxWidgets	0	2,142,713	246,988	xmlnuke	1,623	57,944	1,877
xmakemol	315	18,724	1,380	xqilla	2,534	107,320	9,637
yaml4r	0	10,728	595	zmpp	1,063	15,502	3600

Table 6. Summary of SLOCs and Commits for the two samples

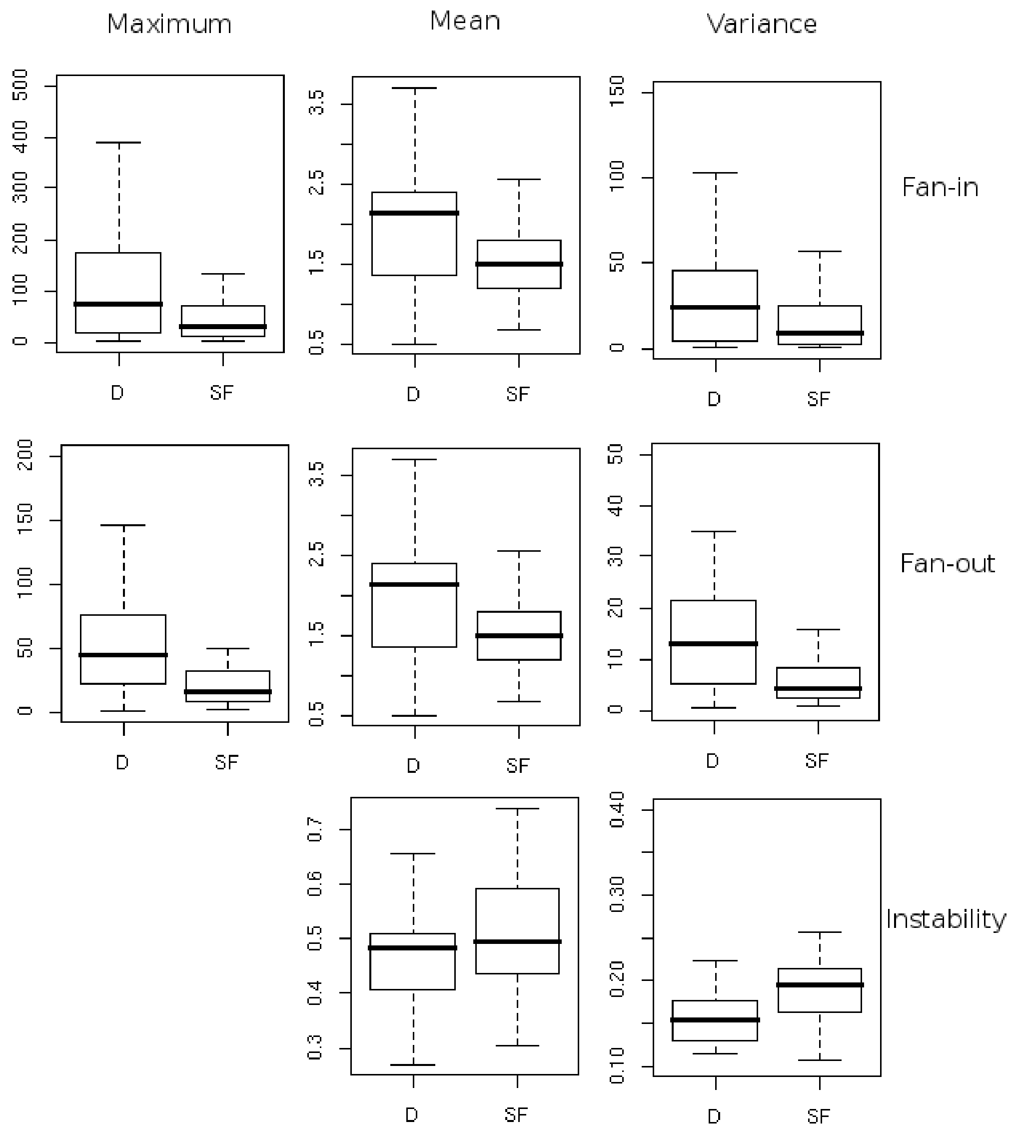

Figure 3. Boxplots of all measured attributes