Quantitatively Exploring Non-code Software Artifacts

Luca Bigliardi,[1] Michele Lanza,[1] Alberto Bacchelli,[2] Marco D'Ambros[1], Andrea Mocci[1]
[1]Faculty of Informatics - University of Lugano, Switzerland
[2]Delft University of Technology, The Netherlands

***Abstract*—Most software engineering research focuses its analyses on source code, because correct, well designed, and efficient program code is the desired end output of software development. Nevertheless, source code is not the only constituent of software systems: Programs also comprise other types of artifacts, such as documentation, build system and configuration files, and graphics. These non-code artifacts only recently got the attention of researchers and are not yet investigated as a whole, but separately and with very specific aims.**

By taking a quantitative perspective, we look into non-code software artifacts to measure their role in software systems. We analyze 35 mature open-source software systems and we address exploratory questions such as: How many non-code software artifacts do software systems contain? How do they relate to source code? How much effort is put into producing and maintaining them? Our results show that a significant portion of systems is made of non-code artifacts, and that programmers spend a relevant part of their effort on non-code artifacts during the development process. Our analysis opens questions for future investigations.

I. Introduction

A major role in software analysis and software quality is played by reverse engineering, defined by Chikofsky and Cross as "*the process of analyzing a subject system to (i) identify the system's components and their interrelationships and (ii) create representations of the system in another form or at a higher level of abstraction*" [1]. This definition, which laid the foundations for a whole research area, stays silent about what should be understood by "subject system." While this may seem a trivial detail, left as an exercise to the reader, it is surprisingly hard to pinpoint what exactly the focus of software analysis should be. The main choice made so far is to focus on source code: the end product of software development. Source code, however, is only one of the facets of a system: Software systems also include a large amount of information and artifacts other than code, such as documentation and examples, build system and configurations, or graphics and translations.

For this reason, recently researchers have started investigating non-code artifacts revolving around source code. For example, many researchers considered the traces left by the development process as valuable non-code artifacts to mine and use for subsequent analyses. They analyzed, for instance, traces in email communication (*e.g.,* [2], [3], [4]), technical forum discussions (*e.g.,* [5], and peer code reviews (*e.g.,* [6]) and have shown their value. Non-code artifacts included together with source code in configuration management system repositories have attracted less interest. In particular, most of the work in this case was focused only on build files: For example, researchers investigated build files and systems both by considering them as if they were software systems of their own, and by studying their co-evolution with source code (*e.g.,* [7], [8]).

Nevertheless, by looking at the files included in code repositories of widespread software applications, it is no wonder that build system files cannot be the only source of differentiation from traditional source code files. For example, we looked into the 4,376 files in the directory of ArgoUML 0.32.2, and we found that only 2,190 (roughly 50%) of them are source files written in Java (*i.e.,* the language ArgoUML is developed in). The remaining files are *not* source code, yet they account for 50% of the directory contents. While we already know that some of these artifacts are build files necessary to obtain the ready-to-use version of the system (*i.e.,* binaries), we know little about the others. This situation raises questions, such as: What are these files and what is their purpose? How much effort does their maintenance require? To which extent are they relevant for program comprehension?

Answering these questions can provide insight for both researchers and practitioners. We mainly focus on the former ones, that is, we motivate why researchers should focus on the challenges offered by the analysis of non-code files. Even if outside the scope of this paper, developers and managers may potentially use empirical evidence to make informed decisions about resource allocation and project planning.

We present a quantitative study that makes a first step into answering these questions. We measure the role of non-code artifacts in software systems by investigating them in 35 open source software (OSS) systems. In particular, we expand our current knowledge of non-build related non-code files. Our results show that, on average, almost a half of the files in each project are non-code artifacts, regardless of the domain, size, or development language. Interestingly, one third of all the commits that developers do involves non-code artifacts, thus showing that this type of artifacts requires considerable effort. This despite the fact that a large portion of non-code artifacts does not evolve after they appear in the repository. Finally, code and non-code artifacts co-evolve steadily for the whole projects' lifetime, thus adding evidence to the existence of a strong link between the two types of artifacts.

Structure of the paper. In Section II we contextualize our work within the related research. In Section III we quantify the non-code artifacts in the last release of the considered OSS projects and study them in relation to software systems' size, domains, and programming languages. In Section IV we study the evolution of non-code artifacts and the attracted effort. In Section V we investigate the relation between code and non-code artifacts. In Section VI we reflect on the findings and discuss directions for future research. Section VII concludes.

II. Related Work

Researchers have been investigating the process behind software development for decades [9], [10], [11], for example to understand the reasons of failures and to suggest ways to improve software quality [12]. The work on these topics is recently fueled by the availability of large amounts of recorded data produced during software development. The mining software repositories (MSR) [13], [14] research field focuses on this data to support program comprehension, development, evolution, as well as to empirically validate common wisdom (*e.g.,* the presence of design flaws in a software system has a negative impact on the quality of the software [15]), or novel ideas (*e.g.,* are popular classes more defect prone? [16]). The MSR field includes work on software engineering aspects that are beyond the source code. For example, research groups analyzed metadata provided by SCM systems to study the impact of software tools on the development process [17], or to explore human factors concerning developers' *territoriality* [18]; Mockus *et al.* studied the quality of a software system as perceived by its users, by mining data captured by project monitoring, tracking infrastructures, and customer support records [19]; Bird *et al.* mined email social networks to investigate common social interaction traits among the participants of open source software projects [2]. Dekhtyar *et al.* suggested that the analysis of software should routinely be augmented with the text that is produced and used during software development [20].

Most research that goes beyond source code focuses on the *process* and the *people* working toward the completion or progress of a software project. Here we address the actual application in terms of non-code software artifacts, *i.e.,* non-code files included in projects' SCM repositories and releases.

Non-code software artifacts in KDE. Robles *et al.* were the first ones to perform a case study on the non-code files included in a versioning system repository [21]. They analyzed the history of the KDE ecosystem, from 1996 to 2006, through its SCM repository. They considered the entire ecosystem as a single unit and grouped files in eight categories (*i.e.,* documentation, images, internationalization, user interface, multimedia, build, development documentation, and code) by means of file extensions. They found that code files account for only 24.4% of the ecosystem. Robles *et al.* analyzed file territoriality, archeology, and the specialization of committers toward certain file types. They found that only 50% of the non-code files were modified within the last two years.

Build system files. Our definition of non-code artifacts includes the files used by build systems to turn "*a set of source code and data files into a suite of executable programs*" [22]. Different researchers investigated build systems: Adams *et al.* devised a framework for understanding and reverse engineering build systems [23] and applied it to analyze the evolution of the Linux kernel build system [7]. They found evidence that the build system evolves, grows in complexity, and must be accordingly maintained [7]. McIntosh *et al.* analyzed the coupling between source code and build system changes to measure the build maintenance overhead on developers [8]. They found that 4–16% of source code work items in Java projects and 27% in C projects require an accompanying build change. This indicates that project managers should "*explicitly account for build maintenance of this magnitude in their project plans and budgets*" [8].

Complementing previous work in this area, which focused on a single system or on a portion of non-code artifacts, we aim at having a bird's-eye view on non-code software artifacts to pinpoint their general traits and discover recurrent patterns.

III. Non-code Software Artifacts

A. Methodology

We base our investigation of non-code software artifacts on 35 mature OSS projects (listed in Table I), written in various languages (mainly C/C++ and Java), developed by unrelated free software communities (*e.g.,* Apache, GNU), and belonging to different domains (*e.g.,* desktop, server, kernel). We take each project's source repository and classify as *code* any file in the project's main language(s) and as *non-code* all the other files. As an assumption, differently from Robles *et al.* (see Section II), who set the build system as a part of code files, we follow the rationale of McIntosh *et al.* [8], who consider them separated from code. Moreover, we do not separate non-code files aimed to developers (*e.g.,* API documentation) or end-users (*e.g.,* build files). Code compilers and interpreters use extensions to recognize source files; similarly, we rely on file extensions to identify code files. We present our analysis and findings in terms of research questions and answers.

B. Non-code at the time of writing

We start our investigation by analyzing the files included in the last release of each system (the fourth column in Table I reports the versions). The focus here is on *size* as an indicator of the impact of non-code. Although a metric based on size could apparently be too imprecise to measure the impact of a feature, size has already been identified as a significant cost factor [24], [25], as a determinant of the difficulty of achieving reliable implementation in a given time [26], and as a valid surrogate for the growth of functional power of a system [27].

How many non-code artifacts do software systems contain?

To answer this question we count the number of files and measure their size. Columns five to eight of Table I report this information. Non-code appears in all the software systems in various degrees: The ratio of non-code varies from less than 5% to more than 95% of the size of a system release and the number of non-code files ranges from less than 3% up to nearly 75% of the total number of files.

The distribution of percentage of non-code files across the systems tends to follow a normal distribution (Shapiro-Wilk normality test [28] failed to reject the normality with a 0.01 significance level) and the most common value is close to 40%.

The bubble graph in Figure 1 depicts an overview of the quantity of non-code in relation to code artifacts. Each system is represented by a bubble, whose area is proportional to the system's size in megabytes and whose color depicts the domain. The *X* axis reflects the percentage of the project's size occupied by non-code, while the *Y* axis reflects the percentage of non-code files over the total number of files.

A considerable diversity appears: While in some systems non-code is barely present (FreeBSD, Linux, Away3D), in

TABLE I. ATTRIBUTES OF THE SOFTWARE SYSTEMS CONSIDERED AS SUBJECTS FOR OUR INVESTIGATION

Name	Domain	Main Language(s)	Last Release					SCM			
			Version	Files		Size (MB)		System	Inception	Number of changes	
				Total	Non-code	Total	Non-code			Total	On Files
argouml	desktop	Java	0.32.2	4376	50.0%	39.59	63.7%	subversion	Jan 26, 1998	17445	16461
audacity	desktop	C/C++	1.3.13-beta	2101	34.9%	35.74	62.8%	subversion	Jan 24, 2010	819	811
eclipse	desktop	Java	3.6.2	35513	42.0%	334.34	55.1%	git	May 9, 2002	NA	NA
gimp	desktop	C	2.6.11	5575	52.7%	110.37	74.0%	git	Jan 1, 1997	27998	27641
inkscape	desktop	C/C++/Python	0.48.1	3225	29.6%	109.38	82.5%	subversion	Oct 30, 2003	17788	17738
jedit	desktop	Java	4.3.2	1356	60.8%	9.90	51.9%	subversion	Sep 2, 2001	6147	6137
pidgin	desktop	C	2.8.0	2113	56.4%	54.75	76.6%	monotone	Mar 23, 2000	NA	NA
vlc	desktop	C/C++	1.1.10	2861	46.4%	125.78	84.9%	git	Aug 8, 1999	43916	43804
vuze	desktop	Java	4605-10	3662	11.3%	28.51	31.7%	subversion	Mar 30, 2010	22585	22508
winscp	desktop	C/C++	4.3.3	1190	12.4%	16.47	13.2%	cvs	Jul 16, 2003	NA	NA
wireshark	desktop	C	1.6.0	4546	42.8%	139.69	25.9%	subversion	Sep 16, 1998	36077	35876
ant	development	Java	1.8.2	2238	46.8%	18.38	54.0%	subversion	Jan 13, 2000	12556	12514
automake	development	Perl	1.11	915	12.6%	4.82	70.7%	git	Mar 13, 1992	4750	4743
away3d	development	ActionScript	4.0.0	185	2.2%	0.89	2.1%	subversion	May 3, 2007	2366	2287
gcc	development	C/Ada	4.6.0	71103	22.3%	408.96	46.9%	subversion	Nov 23, 1988	109591	109529
xulrunner	development	C/C++	2	50114	31.2%	336.96	32.2%	mercurial	Mar 22, 2007	28409	28366
bash	interpreter	C	4.2	1153	40.7%	22.45	81.2%	git	Aug 26, 1996	NA	NA
perl	interpreter	C	5.14.0	5275	28.5%	63.09	53.7%	git	Dec 18, 1987	34820	34715
python	interpreter	C	2.7.2	4248	32.4%	57.26	34.8%	mercurial	Aug 9, 1990	46474	45351
freebsd	kernel	C/Assembly	8.2	8705	15.5%	124.12	3.0%	subversion	Jun 12, 1993	82423	82107
linux	kernel	C/Assembly	2.6.39	36706	14.8%	400.54	7.9%	git	Apr 16, 2005	38632	38630
cvs	scm	C	1.11.23	488	42.8%	10.33	56.6%	cvs	Dec 3, 1994	11298	11295
git	scm	C/Shell/Perl	1.7.5	2137	49.3%	12.05	38.2%	git	Apr 7, 2005	10345	10318
subversion	scm	C	1.6.17	1618	40.0%	42.98	28.3%	subversion	Mar 1, 2000	39194	38878
cassandra	server	Java	0.7.6-2	713	26.5%	20.25	79.5%	subversion	Mar 2, 2009	3126	3076
freenet	server	Java	0.7build01375	4502	75.8%	82.08	90.2%	git	Jan 21, 2005	14475	14472
hibernate	server	Java	3.6.5	13754	64.4%	211.64	91.8%	git	Jun 29, 2007	2653	2626
httpd	server	C	2.2.19	2795	70.8%	33.01	67.7%	subversion	Jul 3, 1996	21652	21571
postgresql	server	C	9.0.4	4941	63.7%	83.37	64.9%	git	Jul 9, 1996	32036	32034
clamav	system	C/C++	0.97.1	2857	37.4%	75.50	71.7%	git	Jul 28, 2003	5778	5769
gnupg	system	C/Assembly	1.4.11	655	40.6%	16.50	72.4%	git	Nov 18, 1997	3224	3223
openssh	system	C	5.8p2	471	24.0%	4.65	41.3%	cvs	Oct 27, 1999	6399	6399
plplot	system	C	5.9.7	2191	58.9%	28.64	73.3%	subversion	May 20, 1992	11771	11715
spamassassin	system	Perl	3.3.1	537	47.3%	4.04	45.5%	subversion	Apr 20, 2001	15702	15682
vim	system	C	7.3	2314	35.9%	35.04	47.9%	mercurial	Jun 13, 2004	2871	2852

others (Hibernate, Freenet) it clearly emerges. In the majority of the cases, software non-code seems to have a non-negligible role. The diversity in the amount of non-code across projects makes us wonder whether software systems may have common traits that are related to the quantity of non-code.

Does the domain relate to the amount of non-code artifacts?

Robles *et al.* stated: "*When we move apart from systems-programming and enter the realm of end-user applications, we find* [non-code] *files*" [21]. Although common sense makes us agree with this statement, no statistical work supports it yet. By analyzing the data and by looking at Figure 1, we spot a relation between the domain and the quantity of non-code. When systems belong to the same domain, the difference in the percentage of non-code files is small: The Linux and FreeBSD kernels differ in the amount of non-code files of only about 1%; SCM systems (*i.e.,* Git, Subversion, CVS) of less than 10%; server-side systems (*i.e.,* PostgreSQL, httpd, and Hibernate) of at most 17%; and language interpreters (*i.e.,* Perl, Python, and Bash) differ of at most 13%. While operating system kernels contain few non-code files, server-side systems show an opposite behavior even though their domain is closer to system-programming than to end-user applications. Vuze and winscp, while being desktop applications, have low amount of non-code files: This is likely due by the fact that they provide a GUI respectively of a peer-to-peer and secure copy protocol. Data provides only partial evidence to the intuition of Robles *et al.* The domain of a project does seem to influence the amount of non-code artifacts, but a finer-grained domain-based categorization of projects would be needed to reveal more.

Does the system's size relate to the amount of non-code?

The size (in terms of bytes and number of files) of the sample systems is *not* related to the amount of non-code artifacts. Large projects, such as Linux, Xulrunner, and Eclipse, are as evenly distributed in Figure 1 as smaller projects, such as WinSCP, OpenSSH, Automake, GnuPG, and jEdit.

Does the language relate to the amount of non-code?

Both JAVA based systems (*e.g.,* Vuze, Eclipse, or Hibernate) and C/C++ based systems (*e.g.,* httpd, Wireshark, or WinScp) do not follow patterns related to the amount of non-code. Therefore, in our sample, the main programming language does not seem to be related to the amount of non-code files, thus we cannot disregard non-code because projects are implemented in a certain paradigm or language.

Reflections. On average, almost a half of the files in each project is non-code. Although we do not know the content of these files, such a high ratio suggests that we should not

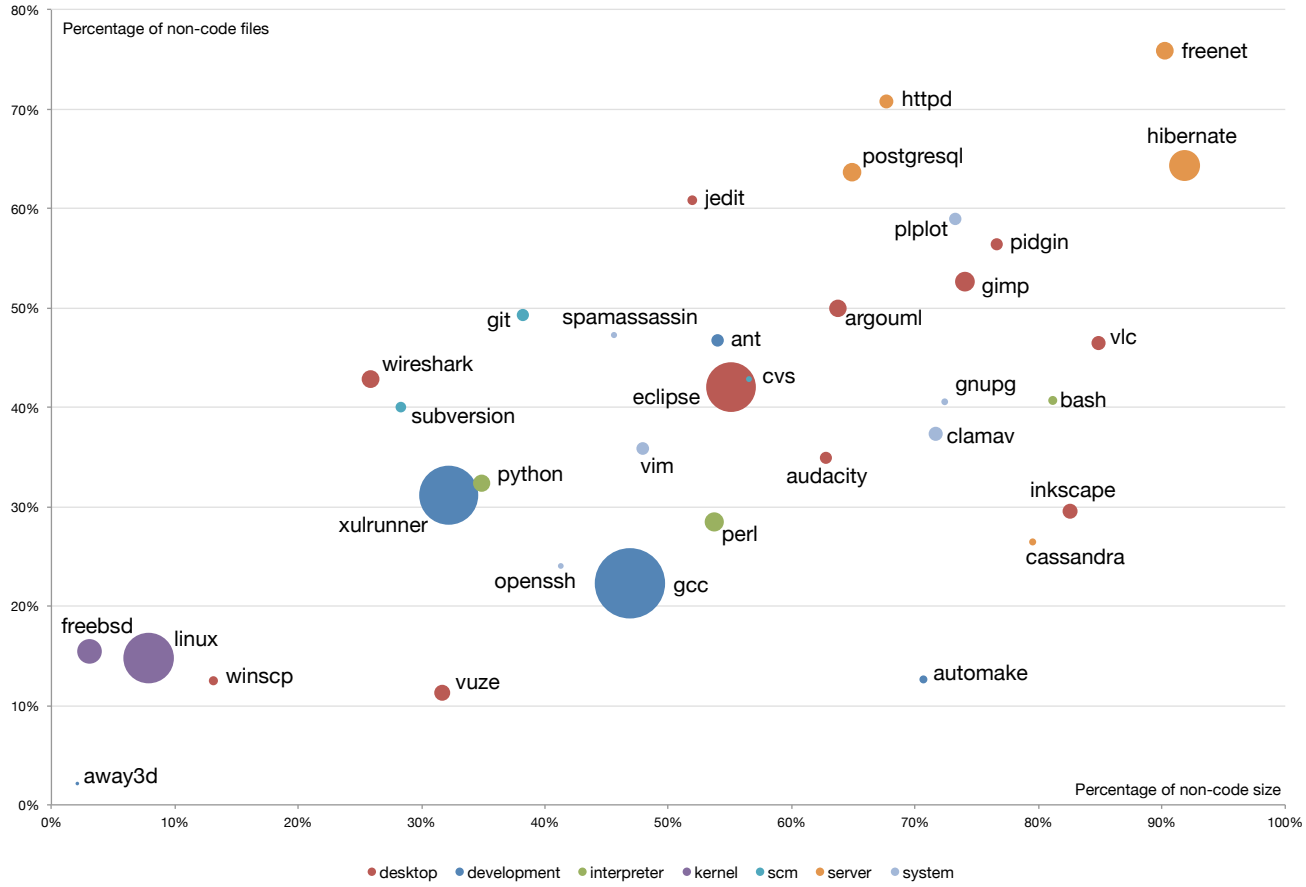

Fig. 1. Percent of non-code in the considered software systems

discard non-code *a priori* from program comprehension and analysis approaches. In particular considering that we cannot rely on domain, size, or language, to predict presence and role of non-code. In the following, we continue our exploration from an evolutionary perspective.

IV. The Evolution of Non-code Software Artifacts

The last columns of Table I regard the source code management (SCM) system. Since the projects use various SCM systems, we imported each repository in a Git repository and implemented a GitPython[1] tool to perform the analysis. Some projects use CVS, which has a granularity that is file-based instead of commit-based. To convert changes from CVS repositories to a commit-based representation we used CVSps.[2] It uses a sliding time window to aggregate in a patchset a group of changes made to a set of files committed together [29].

We study the evolution of 31 projects,[3] starting the analysis from the size of non-code over the project lifetime.

Does the percentage of non-code increase over time?

[1]http://gitorious.org/git-python

[2]http://www.cobite.com/cvsps/

[3]We did not include projects with a small number of commits (Bash, WinSCP), projects with code spread across many repositories (Eclipse), and projects having technical problems during the data importing (Pidgin).

We analyze the Spearman's rank correlation between the time and the evolution of the percentage of non-code files. To obtain the latter we iterate on the commits sorted by time and, whenever a file is added or removed, we calculate a new percentage of non-code files. Table II shows the correlation values, grouped by correlation direction and sorted by decreasing absolute values.

TABLE II. Rank correlation between time and percent of non-code files

Positive Correlation		Negative Correlation		No Correlation	
Name	Value	Name	Value	Name	Value
away3d	0.940	automake	-0.918	postgresql	0.277
wireshark	0.936	hibernate	-0.889	audacity	0.183
freenet	0.920	subversion	-0.823	plplot	0.164
freebsd	0.892	jedit	-0.676	gimp	0.101
gnupg	0.891	xulrunner	-0.606	vim	0.062
ant	0.811	perl	-0.591	inkscape	-0.218
cassandra	0.804	vuze	-0.520	clamav	-0.314
httpd	0.710	linux	-0.512	vlc	-0.317
python	0.637	spamassassin	-0.471	argouml	-0.329
cvs	0.619	openssh	-0.422		
git	0.406				
gcc	0.382				

Since software evolution is a process concerning "*[···] multi-level, multi-loop, multi-agent feedback systems*" [27], many

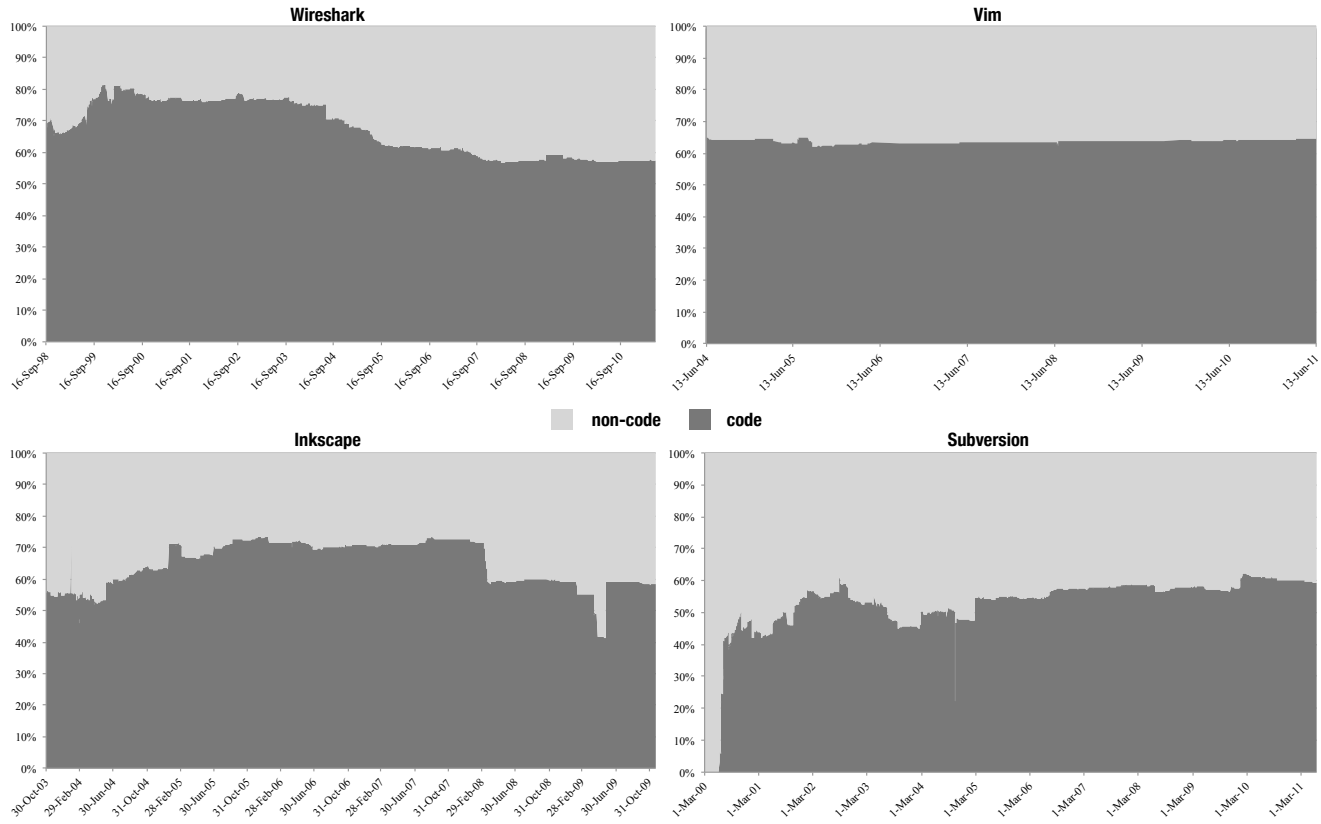

Fig. 2. Different patterns of evolution of non-code artifacts

analyses consider mild correlations (*e.g.,* ca. 0.35) between traits to be valuable. For example, fault prediction studies judge correlations above 0.4 to be strong [30]. We group our results in three sets: positively correlated ($x > 0.35$), negatively correlated ($x < -0.35$), and not correlated ($0.35 < x < -0.35$). Strong correlations ($x > 0.75$) hint at a meaningful relation between a system and the analyzed variables.

The data in Table II shows that there is no general rule: In some projects the percentage of non-code has an increasing trend over time (positive correlation), in others it tends to decrease (negative correlation), and in a third category of systems the percentage of non-code does not vary significantly.

Figure 2 shows stacked percent graphs for representative systems, depicting diverse patterns of non-code evolution. In projects like Freenet and FreeBSD, non-code percentage grows over time; in projects like vim and Gimp, the percentage does not vary significantly; and in projects like Subversion and Hibernate the percentage decreases over time. The ratio between code and non-code files is almost constant in some projects (*e.g.,* vim), but oscillates in others (*e.g.,* Wireshark).

How much effort is spent on non-code maintenance and evolution?

We operationalize the effort spent on non-code maintenance and evolution by measuring the amount of changes involving non-code files. An optimal analysis should quantify the impact of each change, but previous research showed that the number of changes is a fair approximation of the effort, especially when performing large-scale analyses, in which low impact changes are mediated by high impact ones [31].

We consider files as the minimum unit of change for non-code, because there are no tools, yet, that allow for a finer-grained classification of the unstructured nature of some non-code files (*e.g.,* free text, images, or binary files). To measure the effort spent into non-code maintenance and evolution, we measure the percentage of commits that change the content of at least one non-code file over all the commits changing files: We name these *non-code commits*. Further, we name *hybrid* the commits changing at least both one non-code file and one code file. Hybrid commits are a subset of non-code commits: By subtracting the former from the latter, we obtain the commits changing only non-code files (*pure non-code commits*). Analogously, we subtract hybrid commits from commits changing at least one code file (*code commits*) to obtain the commits changing only code files (*pure code commits*).

The mean and the median of non-code commits across the projects are 52.20% and 47.33%, respectively. There is, however, a large difference among projects: We find 9.44% non-code commits in Freenet, while 96.41% in OpenSSH. Table III reports the sorted percentage of non-code commits.

On the 31 considered projects, 14 have more than 50% non-code commits, and 24 have more than 33.3%. The lowest percentage of non-code commits is ca. 10%, meaning that in all the projects at least one commit every 10 includes changes to

TABLE III. PERCENTAGE OF CODE AND NON-CODE COMMITS FOR ALL THE PROJECTS

Name	Types of commits				
	Code		Non-code		Hybrid
	Mixed	Pure	Mixed	Pure	
openssh	71.4%	3.6%	96.4%	28.6%	67.8%
gnupg	77.2%	3.7%	96.3%	22.8%	73.5%
automake	50.2%	5.0%	95.0%	49.8%	45.2%
gimp	67.3%	10.6%	89.4%	32.7%	56.7%
gcc	76.6%	15.4%	84.5%	23.3%	61.2%
cvs	62.9%	20.0%	80.0%	37.1%	42.9%
spamassassin	30.6%	22.0%	78.0%	69.4%	8.5%
clamav	81.8%	24.6%	75.4%	18.2%	57.3%
jedit	73.9%	29.5%	70.5%	26.1%	44.4%
postgresql	55.1%	38.1%	62.0%	44.9%	17.1%
httpd	49.8%	38.5%	61.5%	50.2%	11.4%
plplot	48.8%	39.8%	60.2%	51.3%	9.0%
ant	64.6%	44.0%	56.0%	35.4%	20.6%
vim	87.8%	48.4%	51.6%	12.2%	39.4%
cassandra	80.9%	51.3%	48.7%	19.2%	29.5%
perl	75.6%	52.7%	47.3%	24.4%	22.9%
hibernate	68.7%	53.6%	46.4%	31.3%	15.1%
xulrunner	79.8%	54.0%	45.9%	20.1%	25.7%
git	74.9%	54.4%	45.6%	25.1%	20.4%
python	68.5%	56.4%	43.6%	31.5%	12.2%
inkscape	73.5%	60.5%	39.6%	26.5%	13.1%
argouml	70.5%	65.1%	34.9%	29.5%	5.4%
subversion	71.4%	65.2%	34.8%	28.6%	6.2%
vlc	76.3%	65.4%	34.6%	23.8%	10.9%
wireshark	81.7%	68.1%	31.9%	18.3%	13.7%
audacity	77.4%	73.9%	26.1%	22.6%	3.6%
linux	92.2%	75.3%	24.7%	7.8%	16.9%
vuze	85.9%	78.0%	22.0%	14.2%	7.9%
freebsd	89.8%	84.5%	15.5%	10.3%	5.3%
away3d	95.8%	89.7%	10.3%	4.2%	6.0%
freenet	94.4%	90.6%	9.4%	5.6%	3.9%

non-code files. For all projects but seven, this holds for at least one commit every three. The data concerning pure non-code commits shows that 10 projects have more than 30% of pure non-code commits, 22 projects have more than 20%, and all but three have more than 10%.

The results indicate that developers devote a significant effort to maintain and evolve non-code artifacts: For most projects, a third of the commits involves non-code files and a tenth involves non-code files *only*.

Does the maintenance effort on non-code increase as a system evolves?

We analyze the rank correlation between project age (*i.e.,* time) and evolution of the percentage of non-code commits. To obtain the evolution of these percentages, our tool iterates on all the commits from the oldest to the most recent, and updates the number of commits and percentages whenever a commit changes or removes a file. Table IV reports the results.

Different projects show diverse behaviors and data does not support any general rule: Some projects (*e.g.,* Cassandra, SpamAssassin) show a strong positive correlation, meaning that the effort increases over time; other projects (*e.g.,* jEdit, Subversion) show the opposite behavior, *i.e.,* the negative correlations indicate that the effort spent on non-code files decreases with time. In a third category of projects, the effort spent on non-code files does not relate with time.

TABLE IV. RANK CORRELATION: TIME VS. PERCENT OF NON-CODE COMMITS

Positive Correlation		Negative Correlation		No Correlation	
Name	Value	Name	Value	Name	Value
cassandra	0.842	jedit	-0.801	httpd	0.301
spamassassin	0.717	subversion	-0.647	freenet	0.217
gcc	0.694	audacity	-0.600	gnupg	0.184
openssh	0.631	hibernate	-0.447	cvs	0.126
automake	0.570	vuze	-0.426	away3d	0.112
git	0.513	freebsd	-0.412	xulrunner	0.082
linux	0.513	argouml	-0.394	perl	0.078
python	0.425	vim	-0.356	postgresql	0.004
ant	0.370	clamav	-0.355	wireshark	-0.108
plplot	0.350			inkscape	-0.238
				gimp	-0.302
				vlc	-0.312

How is the non-code effort distributed over the evolution of systems?

To answer this question we use the Gini coefficient, an indicator originally used to study the distribution of wealth in a population [32]. To define this coefficient, we introduce the Lorenz curve [33]: an indicator of inequality within a population. The curve is plotted putting on the Y axis the proportion of the distribution assumed by the bottom X% of the population. The use of proportions in the curve allows comparisons between distributions of different data types.

To build the Lorenz curve for a given project, we perform the following steps: (i) Consider all the commits over the project's history and group them in 20-commit bins[4], (ii) compute the percentage of non-code commits in each bin, (iii) sort the bins according to the increasing percentages of non-code commits, and (iv) plot the Lorenz curve, where the X coordinate of a point is a bin, and the Y coordinate is the normalized cumulative sum[5] of percentage of non-code commits up to that point. Figure 3 shows the Lorenz curve for httpd, where the bisector indicates perfect equality.

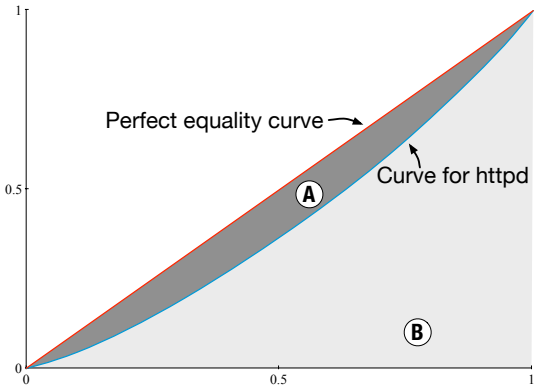

Fig. 3. Lorenz curve for the httpd project

[4]We grouped changes using a commit-based division, instead of time-based, because OSS projects are not developed at a constant pace.

[5]We normalize the cumulative sum, so that when all bins are considered (rightmost point of the curve) the total cumulative percentage is 1.

We calculate the Gini coefficient as $A \cdot (A+B)^{-1}$, where A and B are the areas shown in Figure 3 between the equality bisector and the Lorenz curve (A) and under the Lorenz curve (B). The Gini coefficient is bounded between 0 and 1, where 0 represents perfect equality (*i.e.,* all the bins account for the same percentage of non-code commits, *i.e.,* the bisector line in Figure 3) and 1 represents total inequality (*i.e.,* all the non-code commits are concentrated in a single bin). We opted for the Gini coefficient because other statistic tools such as the skew (which measures data asymmetry) and the kurtosis (which measures data peakedness) are unbounded, whereas the Gini coefficient is bounded, thus it can be used to compare different projects. Moreover, it handles populations with different sizes (our projects have different lifetime spans) and makes no assumptions on the data distribution.

TABLE V. GINI COEFFICIENTS FOR ALL PROJECTS

Name	Gini coeff.	Name	Gini coeff.	Name	Gini coeff.
away3d	0.396	plplot	0.183	git	0.111
freenet	0.352	vlc	0.173	clamav	0.107
audacity	0.300	cassandra	0.169	perl	0.106
subversion	0.268	hibernate	0.161	spamassassin	0.103
argouml	0.225	wireshark	0.154	gimp	0.085
freebsd	0.209	httpd	0.150	postgresql	0.085
vim	0.203	ant	0.140	cvs	0.078
vuze	0.196	jedit	0.140	automake	0.041
linux	0.194	xulrunner	0.126	gnupg	0.022
inkscape	0.192	gcc	0.124	openssh	0.018
python	0.188				

Table V reports the Gini coefficients for all projects: All values are below 0.4, indicating fair distributions of the percentages of non-code commits. Different systems exhibit different behaviors: For example, OpenSSH and GnuPG show an almost perfect equality in the distribution of non-code commits; other systems, such as Away3D and Freenet, have a more unbalanced behavior. These results indicate that the effort spent on non-code artifacts is not focused in a few intense periods, but it is spread over the entire evolution.

Does the percentage of non-code relate to the effort required to maintain it?

We investigate whether the evolution of the percentages of non-code files and non-code commits are correlated. Table VI presents the rank correlation between the evolution of the two percentages across all the projects.

The mean and the median are 0.034 and 0.033 respectively. In the majority of the projects the correlation varies between 0.3 and -0.3, suggesting that an increased amount of non-code over time does not affect the effort over time; this might indicate that parts of non-code, once added to the project, are not evolving.

Do portions of non-code not change?

We study the time passed since a certain percentage of non-code has changed in the system. To calculate the percentage of non-code that does not evolve we consider each non-code file in the last revision of a project and calculate how many days passed between its last change and the last revision of the project. Table VII reports how many days passed since the last change of various percentages of non-code files.

TABLE VI. RANK CORRELATION: PERCENTAGE OF NON-CODE FILES VS. COMMITS

Positive Correlation		No Correlation			
Name	Value	Name	Value	Name	Value
gcc	0.634	openssh	0.343	postgresql	0.016
cassandra	0.585	plplot	0.307	wireshark	-0.065
spamassassin	0.572	python	0.287	audacity	-0.081
automake	0.372	git	0.274	inkscape	-0.139
		ant	0.238	vlc	-0.196
Negative Correlation		linux	0.217	freebsd	-0.207
Name	**Value**	httpd	0.192	clamav	-0.232
jedit	-0.651	gnupg	0.118	hibernate	-0.249
subversion	-0.485	cvs	0.101	argouml	-0.260
		freenet	0.094	vuze	-0.261
		away3d	0.068	gimp	-0.293
		xulrunner	0.033	vim	-0.300
		perl	0.031		

TABLE VII. DAYS PASSED SINCE A PERCENTAGE NON-CODE FILES DID NOT CHANGE

Name	Last Change (days)			Name	Last Change (days)		
	80%	50%	20%		80%	50%	20%
cvs	1005	2047	2431	git	85	563	1352
gcc	293	1419	1618	subversion	215	554	855
gimp	261	1354	2046	xulrunner	84	527	1081
argouml	671	1330	3462	vlc	121	521	1170
gnupg	132	1289	1780	vuze	215	514	1063
jedit	524	1143	2507	audacity	486	486	486
vim	302	1083	2555	linux	82	314	1037
ant	74	962	1175	clamav	228	228	488
openssh	29	942	3311	python	73	225	636
freebsd	161	933	2332	postgresql	60	204	713
spamassassin	457	737	2429	httpd	18	184	1191
plplot	156	685	1818	hibernate	82	164	246
inkscape	205	654	845	cassandra	66	127	319
away3d	239	620	655	automake	59	59	1243
wireshark	202	611	1340	freenet	43	43	409
perl	124	595	628				

We notice different change patterns among the systems: In projects like Hibernate or Cassandra almost all the non-code evolves, only less than 20% of non-code files did not change in the last year of development; in projects like CVS or ArgoUML almost all the non-code does not evolve (*i.e.,* 80% of non-code files did not change in the last 2–3 years).

Reflections. Evolutionary data uncovers two traits of software non-code. On the one side, results tell us that developers do work on non-code artifacts: One third of all the commits involves non-code files, with a steady effort spent through the entire life of projects. On the other side, the evolution of non-code takes a variety of different paths that depend on the project and cannot be generalized in a single pattern. In other words, each system's non-code has its own story. We also assessed that a large portion of non-code does not evolve. Our investigation proceeds by exploring the relation between the development of code and non-code artifacts.

V. CODE AND NON-CODE

Do developers work on code and non-code contemporarily?

We operationalize the effort spent in maintaining and evolving code and non-code at the same time by analyzing how *pure* and *hybrid* commits (defined in Section IV) relate.

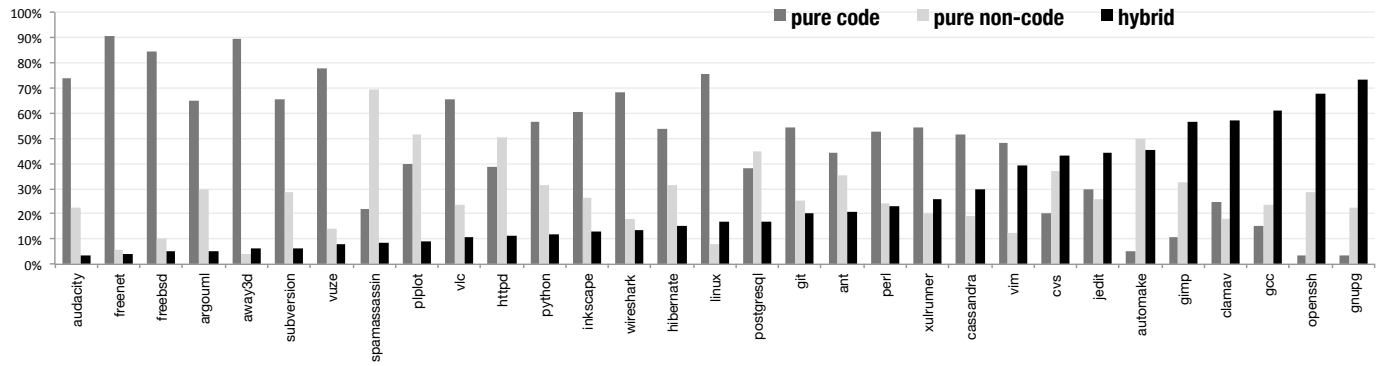

Fig. 4. Amount of commits types in the various projects

Table III shows the percentage of the different commit types. In all the projects, developers work on code and non-code at the same time: The ratio of hybrid commits ranges from 3.6% of Audacity to 73.5% of GnuPG. The mean over all the hybrid commits percentages is 24.9% and the median is 16.8%.

To understand whether there is influence between the different types of work, we measure the rank correlation between the ratio of the various types of commits. The correlation between the ratio of hybrid commits and the ratio of pure code commits is -0.773, thus the more developers work on code and non-code at the same time, the less they work exclusively on code. The relation does not hold on the non-code side: The correlation between the ratio of hybrid commits and the ratio of pure non-code commits is -0.037.

Figure 4 reports the percentage of pure code, pure non-code, and hybrid commits; the projects are ordered by increasing hybrid commits to ease the observation of the behavior of the two correlations just described: The percentages of pure code commits decrease from left to right, while pure non-code commits do not present any pattern.

How is the hybrid effort distributed over the evolution?

Within a single project, the ratio among commit types changes over time. Looking at Figure 5 we see an inception period: ArgoUML and SpamAssassin started with many hybrid commits, while Subversion and httpd started with almost only pure non-code commits. A manual inspection of the repositories revealed that Subversion developers started using the repository mainly to store project notes and that httpd developers have been using the repository for many years just to store documentation.

To understand whether periods of high concurrent activity on code and non-code recur in the evolution of a project, we calculated the Gini coefficients for the distributions of hybrid commits percentage (Table VIII) . For the majority of the projects the coefficient is below 0.4, indicating that in general hybrid commits are equally distributed over time.

The spread of hybrid commits over a system evolution indicates that developers not only spend a significant effort on non-code (as previously discussed in Section IV), but also constantly work on code and non-code at the same time.

TABLE VIII. GINI COEFFICIENTS FOR HYBRID COMMITS

Name	Gini coeff.	Name	Gini coeff.
freenet	0.4792	vuze	0.2142
spamassassin	0.4672	hibernate	0.2123
python	0.4438	postgresql	0.2098
away3d	0.4324	automake	0.2043
audacity	0.3867	wireshark	0.1997
plplot	0.3602	cvs	0.1946
argouml	0.3201	ant	0.1835
inkscape	0.3183	gcc	0.1814
jedit	0.3032	git	0.1791
vlc	0.2993	perl	0.1740
subversion	0.2907	gimp	0.1691
linux	0.2723	xulrunner	0.1650
vim	0.2613	clamav	0.1405
httpd	0.2482	openssh	0.0702
freebsd	0.2370	gnupg	0.0496
cassandra	0.2241		

Do specific parts of non-code co-evolve with code?

To answer this question we investigate whether some specific non-code files appear in hybrid commits more often than others. For each file, we compute non-code likelihood to appear in a hybrid commit with the following formula:

$$likelihood = \frac{\#\ of\ hybrid\ commits}{\#\ of\ non-code\ commits} \cdot \#\ of\ hybrid\ commits$$

Using this formula we measure the ratio between the number of hybrid commits and non-code commits changing a certain file. We give prominence to files participating in a high number of commits scaling the ratio with the number of hybrid commits. The likelihood tends to zero for non-code files that change frequently but independently from hybrid commits. In all the projects only a small fraction of non-code files co-evolve frequently with code and the distribution of likelihood is well approximated using the power law.

Reflections. Code and non-code artifacts seem to be intertwined, hence they do not only co-exist, but they also co-evolve steadily for the whole project's lifetime. This suggests the existence of a strong link between code and non-code, which future studies might investigate to see how it can be used to

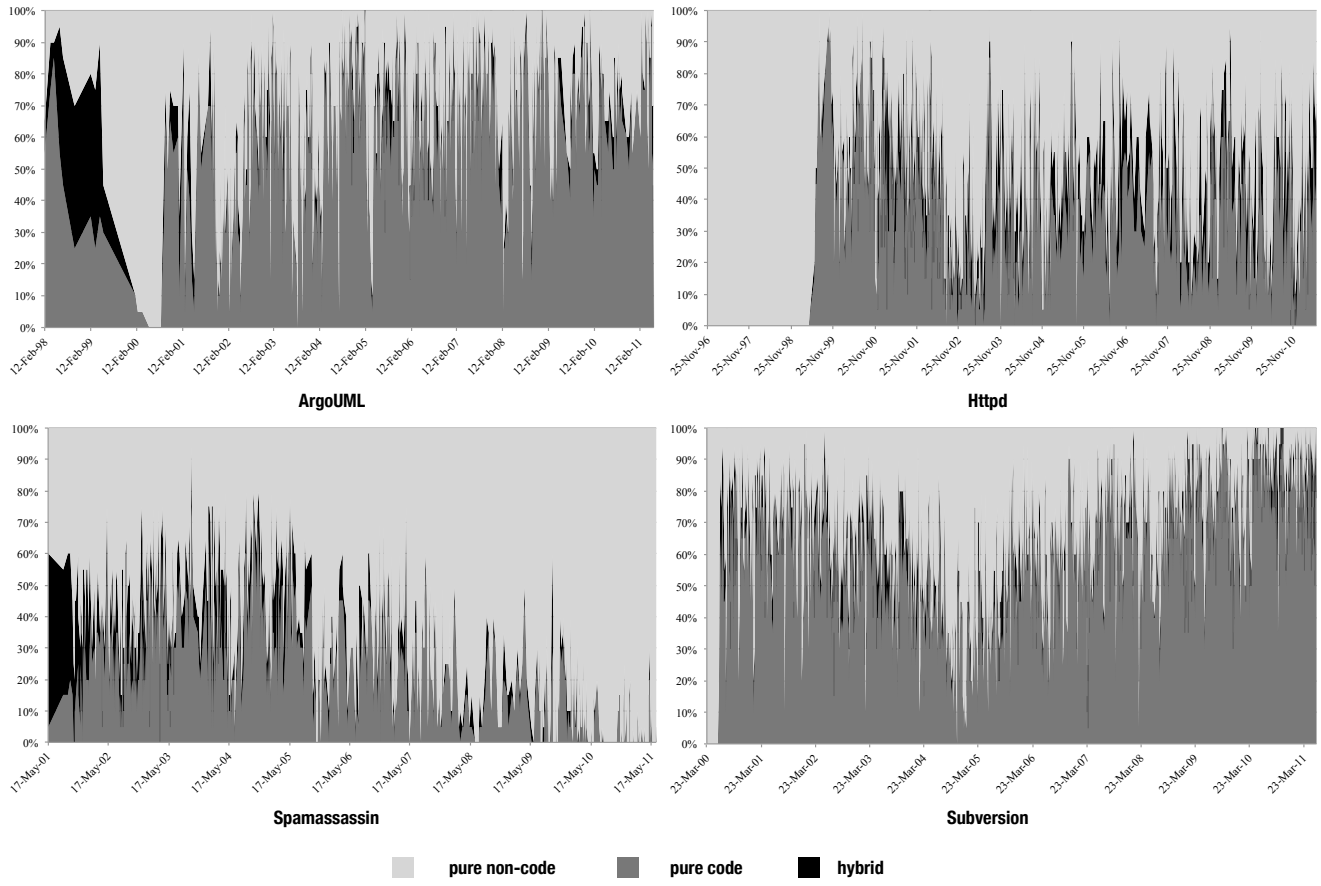

Fig. 5. Percentage variations in the commits types

improve software quality, and analysis techniques. Moreover, the presence of a kernel of non-code files that co-evolve with code mirrors the behavior of code artifacts reported by Vasa *et al.* [34]. They found that in a system "*of the classes that have changed, most have been touched a few times, only a small proportion is modified several times*" [34]. Since Vasa *et al.* also found that "*classes with the highest rates of change tend to be the most popular*" [35], future studies might investigate whether this concept holds for non-code and what determines the relevance of certain non-code files.

VI. Discussion

The results emerged from our quantitative study suggests that non-code artifacts have a not neglectful role in the development of software systems. We found that non-code constitutes a significant part of software systems, in terms of size and file number, regardless of the domain, programming language, and size. Moreover, its ratio often grows over time.

Trying to triangulate from another perspective the importance of non-code, we explored the histories of projects. We found that programmers conduct a steady activity on non-code, spread over the entire evolution: Non-code artifacts' maintenance and evolution have a weighty impact on the overall development effort. The percentage of commits that includes both code and non-code files (*i.e.,* hybrid commits) across all the considered systems ranges from 3.6% up to 73.5%, with an average of 23.7% (see last column in Table III). By counting the percentage of all commits involving non-code (*i.e.,* we also consider commits with only non-code files), we found that the average raises to 51.3% (4th column in Table III), thus showing a significant amount of work around the entire non-code files.

The analysis of non-code evolution generated more questions than answers. First of all, we verified that there is no clear pattern that can be generalized to all the considered projects. For example, the ratio of effort on non-code shows conflicting growth trends among the different systems (see Table II). In addition, results tell us that in most systems 50% of the non-code files has not evolved in the last year. This confirms the findings of Robles *et al.*, who measured that 50% of the KDE ecosystem files were not modified in the last two years. Finally, by investigating the changes affecting code and non-code concurrently, we found that the likelihood of non-code files to be part of hybrid changes follows a power law distribution, meaning that there is a small and well defined non-code kernel co-evolving with code. However, the role of this kernel is to be clarified: Even though as a first hypothesis we supposed this non-code kernel to be related to the building system, by analyzing the Linux system we found that non-code files mainly co-evolving with code are *not* build related, but are documentation related. This overall knowledge on non-code hints at the importance of further research to develop new theories to clarify the role of non-code and its evolution.

VII. Conclusions

By conducting a quantitative analysis on 35 open source system projects, we found that non-code artifacts play an important role. Not only does non-code constitute a significant portion of systems, but also developers spend considerable effort on maintaining and evolving it. Under this light, it is reasonable to suggest that non-code should be subject of further investigation to achieve a holistic view of how software systems evolve, and to gain a more detailed insight in the many facets composing the activity of the developers. In the future we plan to (i) investigate how non-code can be classified in categories to perform a comparative analysis of their content and a fine-grained analysis of their evolution; (ii) analyze the structural links between code and non-code, by examining their co-evolution and their content; (iii) understand if developers producing code and non-code artifacts are the same, and if they are not how they interact; (iv) make sense of the reason why most non-code ages without changes and comprehend the impact of age on non-code popularity and value; and (v) understand the role and categories that compose the kernel of non-code that continuously changes with code.

Acknowledgements. We acknowledge the financial support of the Swiss National Science foundation through project No. 153129 "ESSENTIALS".

References

[1] E. Chikofsky and J. Cross, "Reverse engineering and design recovery: A taxonomy," *IEEE Software*, vol. 7, no. 1, pp. 13–17, Jan. 1990.

[2] C. Bird, D. Pattison, R. D'Souza, V. Filkov, and P. Devanbu, "Latent social structure in open source projects," in *Proc. FSE 2008 (16th Int'l Symp. on Foundations of Software Engineering)*. ACM, 2008, pp. 24–35.

[3] A. Bacchelli, M. Lanza, and V. Humpa, "RTFM (Read The Factual Mails) –augmenting program comprehension with remail," in *Proceedings of CSMR 2011 (15th IEEE European Conference on Software Maintenance and Reengineering)*, 2011, pp. 15–24.

[4] A. Guzzi, A. Bacchelli, M. Lanza, M. Pinzger, and A. van Deursen, "Communication in open source software development mailing lists," in *Proceedings of MSR 2013 (10th IEEE Working Conference on Mining Software Repositories)*, 2013, p. to be published.

[5] L. Ponzanelli, A. Bacchelli, and M. Lanza, "Leveraging crowd knowledge for software comprehension and development," in *Proceedings of CSMR 2013 (17th European Conference on Software Maintenance and Reengineering)*. IEEE CS Press, 2013, pp. 57–66.

[6] P. C. Rigby and C. Bird, "Convergent contemporary software peer review practices," in *Proceedings of ESEC/FSE 2013 (9th Joint Meeting on Foundations of Software Engineering)*, ser. ESEC/FSE 2013. ACM, 2013, pp. 202–212.

[7] B. Adams, H. Tromp, K. D. Schutter, and W. D. Meuter, "Design recovery and maintenance of build systems," in *Int'l Conf. on Software Maintenance*, 2007, pp. 114–123.

[8] S. McIntosh, B. Adams, T. H. Nguyen, Y. Kamei, and A. E. Hassan, "An empirical study of build maintenance effort," in *Proc. of ICSE 2011 (33th Int'l Conf. on Software Engineering)*. ACM, 2011, pp. 141–150.

[9] M. E. Conway, "How do committees invent?" *Datamation*, vol. 14, no. 4, pp. 28–31, 1968.

[10] F. Brooks, *The Mythical Man-Month*, 2nd ed. Addison-Wesley, 1995.

[11] T. D. Marco, *Peopleware - Productive Projects and Teams*. Dorset House, 1999.

[12] N. Nagappan, B. Murphy, and V. Basili, "The influence of organizational structure on software quality: an empirical case study," in *Proceedings of ICSE '08 (30th international conference on Software engineering)*. ACM, 2008, pp. 521–530.

[13] H. H. Kagdi, M. L. Collard, and J. I. Maletic, "A survey and taxonomy of approaches for mining software repositories in the context of software evolution," *Journal of Software Maintenance*, vol. 19, no. 2, pp. 77–131, 2007.

[14] A. E. Hassan, "The road ahead for Mining Software Repositories," in *Proc. of FoSM 2008 (Frontiers of Software Maintenance)*, 2008, pp. 48–57.

[15] M. D'Ambros, A. Bacchelli, and M. Lanza, "On the impact of design flaws on software defects," in *Proceedings of QSIC 2010 (10th International Conference on Quality Software)*. IEEE CS Press, 2010, pp. 23–31.

[16] A. Bacchelli, M. D'Ambros, and M. Lanza, "Are popular classes more defect prone?" in *Proc. of FASE 2010 (13th Int'l Conf. on Fundamental Approaches to Software Engineering)*, 2010, pp. 59–73.

[17] D. Atkins, T. Ball, T. Graves, and A. Mockus, "Using version control data to evaluate the impact of software tools: A case study of the version editor," *IEEE Trans. on Software Engineering*, vol. 28, no. 5, pp. 625–637, May 2002.

[18] D. M. German, "Using software trails to reconstruct the evolution of software," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 16, no. 6, pp. 367–384, 2004.

[19] A. Mockus, P. Zhang, and P. L. Li, "Predictors of customer perceived software quality," in *Proc. of ICSE 2005 (27th Int'l Conf. on Software engineering)*. ACM, 2005, pp. 225–233.

[20] A. Dekhtyar, J. H. Hayes, and T. Menzies, "Text is software too," in *Proc. of MSR 2004 (1st Int'l Workshop on Mining Software Repositories)*, 2004, pp. 22–26.

[21] G. Robles, J. M. González-barahona, and J. J. M. Guervós, "Beyond source code: The importance of other artifacts in software development (a case study)," *Journal of Systems and Software*, vol. 79, pp. 1233–1248, 2006.

[22] B. Adams, "Co-evolution of source code and the build system," in *Proc. of ICSM 2009 (25th Int'l Conf. on Software Maintenance)*. IEEE, 2009.

[23] B. Adams, K. D. Schutter, H. Tromp, and W. D. Meuter, "The evolution of the linux build system," *ECEASST*, 2007.

[24] B. W. Boehm, *Software Engineering Economics*. Prentice Hall, 1981.

[25] ——, *Software Cost Estimation with Cocomo II*. Prentice Hall, 2000.

[26] M. Lehman, J. Ramil, and U. Sandler, "An approach to modelling long-term growth trends in software systems," in *Proc. of ICSM 2001 (17th Int'l Conf. on Software Maintenance)*. IEEE Computer Society, 2001, pp. 219–228.

[27] M. M. Lehman, J. F. Ramil, P. D. Wernick, D. E. Perry, and W. M. Turski, "Metrics and laws of software evolution - the nineties view," in *Proc. of the 4th Int'l Symp. on Software Metrics*. IEEE Computer Society, 1997, pp. 20–32.

[28] S. S. Shapiro and M. B. Wilk, "An analysis of variance test for normality (complete samples)," *Biometrika*, vol. 3, no. 52, 1965.

[29] T. Zimmermann and P. Weißgerber, "Preprocessing CVS data for fine-grained analysis," in *Proc. of the First Int'l Workshop on Mining Software Repositories*, May 2004, pp. 2–6.

[30] T. Zimmermann and N. Nagappan, "Predicting defects using network analysis on dependency graphs," in *Proc. of ICSE 2008 (30th Int'l Conf. on Software Engineering)*, 2008.

[31] T. L. Graves and A. Mockus, "Inferring change effort from configuration management databases," *IEEE Int'l Symp. on Software Metrics*, vol. 0, p. 267, 1998.

[32] R. Vasa, M. Lumpe, P. Branch, and O. Nierstrasz, "Comparative analysis of evolving software systems using the gini coefficient." in *Proceedings of ICSM 2009 (25th IEEE International Conference on Software Maintenance)*. IEEE, 2009, pp. 179–188.

[33] M. O. Lorenz, "Methods of measuring the concentration of wealth," *Publications of the American Statistical Association*, vol. 9, no. 70, pp. 209–219, June 1905.

[34] R. Vasa, J. guy Schneider, O. Nierstrasz, and C. Woodward, "On the resilience of classes to change," *Electronic Communication of The European Association of Software Science and Technology*, 2007.

[35] R. Vasa, J. guy Schneider, and O. Nierstrasz, "The inevitable stability of software change," in *Proc. of ICSM 2007 (23rd Int'l Conf. on Software Maintenance)*, 2007, pp. 4–13.