Preserving Knowledge in Software Projects

Omar Alam[a], Bram Adams[b,*], Ahmed E. Hassan[c]

[a]*SEL, School of Computer Science, McGill University, Canada*
[b]*MCIS, Département de Génie Informatique et Génie Logiciel, École Polytechnique de Montréal, Canada*
[c]*SAIL, School of Computing, Queen's University, Canada*

Abstract

Up-to-date preservation of project knowledge like developer communication and design documents is essential for the successful evolution of software systems. Ideally, all knowledge should be preserved, but since projects only have limited resources, and software systems continuously grow in scope and complexity, one needs to prioritize the subsystems and development periods for which knowledge preservation is more urgent. For example, core subsystems on which the majority of other subsystems build are obviously prime candidates for preservation, yet if these subsystems change continuously, picking a development period to start knowledge preservation and to maintain knowledge for over time become very hard. This paper exploits the time dependence between code changes to automatically determine for which subsystems and development periods of a software project knowledge preservation would be most valuable. A case study on two large open source projects (PostgreSQL and FreeBSD) shows that the most valuable subsystems to preserve knowledge for are large core subsystems. However, the majority of these subsystems (1) are continuously foundational, i.e., ideally for each development period knowledge should be preserved, and (2) experience substantial changes, i.e., preserving knowledge requires substantial effort.

Keywords: software maintenance; documentation; knowledge preservation; empirical analysis; mining software repositories

1. Introduction

The global scale of today's software development makes it very easy for project teams to lose track of the context and knowledge about their systems, such as best practices and design rationale [1, 2, 3]. Even for those projects that have preserved system knowledge in the form of code comments [1], design documentation, manuals, tutorials, comprehensive test suites, dedicated system experts, developer training [4] and/or archives of relevant development artifacts, it is very challenging to keep this

*Corresponding author
Email addresses: omar.alam@mail.mcgill.ca (Omar Alam), bram.adams@polymtl.ca (Bram Adams), ahmed@cs.queensu.ca (Ahmed E. Hassan)

knowledge up-to-date [3]. This is problematic, since lack of up-to-date system knowledge has been identified as the second largest root cause of defects in software systems [4]. Preservation of knowledge is also crucial for decision processes and program understanding [5].

The reason why up-to-date system knowledge is often lacking is not just ignorance, but rather that preserving all knowledge simply is infeasible because of the abundance of knowledge that could be preserved. Software systems keep on growing in size and complexity over time [6, 7], and this growth is typically accompanied by a growth in the number of contributors, mailing list discussions and bug reports [8, 9]. Figuring out for which subsystems preservation of knowledge would be the most valuable and sustainable is complicated. Ideally, so-called "foundational" subsystems, i.e., subsystems that have many other subsystems building and depending on their APIs (Application Programming Interfaces), are prime candidates for preservation. Yet, blindly preserving knowledge of every development period of a foundational subsystem is not feasible nor efficient, since for some subsystems all periods contain major changes that force dependent subsystems to change, while for others hardly any period contains major changes (e.g., only internal bug fixes).

To support practitioners in prioritizing subsystems and development periods for which knowledge should be preserved the most, it is important to consider the trade-off between space, time and effort. For example, since HTML rendering is the core business of web browsers, the HTML rendering subsystem is a foundational subsystem on which many other subsystems build (space) and that continuously evolves (time) because of major performance and functionality changes (effort). Other browser subsystems, like the SSL/TLS subsystem, have less subsystems building on them (space), and change less frequently (time) and significantly (effort). The space and time dimensions determine the subsystem and its development periods that are most valuable to preserve knowledge for, whereas the effort dimension determines how much new knowledge potentially needs to be preserved. The HTML rendering subsystem is more valuable to preserve knowledge for than the SSL/TLS subsystem, but requires preservation of the knowledge of almost every development period, which takes significant effort given the many changes. Hence, in some cases practitioners could opt to preserve knowledge for SSL/TLS instead.

To support the analysis of the space, time and effort dimensions of knowledge preservation in real-life software systems, we propose an automated approach to identify foundational subsystems and periods for a project. The approach is based on our earlier work on the time dependence of changes [10, 11]. Time dependence of changes captures for each source code entity E in a particular revision of a software system the specific revision of all source code entities (such as API methods) on which E builds. In this paper, we lift time dependence of changes up from revisions to development periods and from entities to subsystems. This lifted version of time dependence allows us to calculate for each subsystem in each development period:

- the "foundationality" [1], i.e., the degree to which other subsystems build on the

[1] Although the word foundationality does not exist in the English dictionary, it has been used by philosophers and even by some computer scientists .

subsystem;
- the "sporadicity", i.e., how irregularly the foundationality of the subsystem is distributed across development periods;
- the number of source code changes to a subsystem, i.e., how much new knowledge is added (and potentially should be preserved).

We apply our approach on two large open source systems (PostgreSQL and FreeBSD) to address the following three research questions:

Q1 Which subsystems should have a higher priority for knowledge preservation?

> A project develops few highly foundational subsystems that provide the project's core structure and hence should be preserved first.

Q2 Which development periods should have a higher priority for knowledge preservation?

> Most foundational subsystems are continuously foundational, which means that ideally knowledge should be preserved for every development period.

Q3 How much effort is involved in preserving knowledge of foundational subsystems?

> Foundationality of a subsystem in a development period correlates with the number of changes to the subsystem in that period. In other words, a lot of new knowledge is added in foundational development periods, which requires more effort to preserve.

Our work provides practitioners with an automatic approach to help them prioritize which subsystems to preserve knowledge for. However, since the most foundational subsystems turn out to require substantial preservation effort (experience most of the changes), more work is needed on prioritizing the foundational subsystems that need knowledge to be preserved.

Organization of the Paper. The paper is organized as follows. Section 2 presents our methodology based on the time dependence of changes. Section 3 presents the three research questions that we study using our methodology. Section 4 explains the setup of the two case studies that we performed, and discusses the case study results for each research question. Section 5 discusses threats to validity, whereas Section 6 discusses related work. Section 7 summarizes our findings and concludes the paper.

2. Methodology

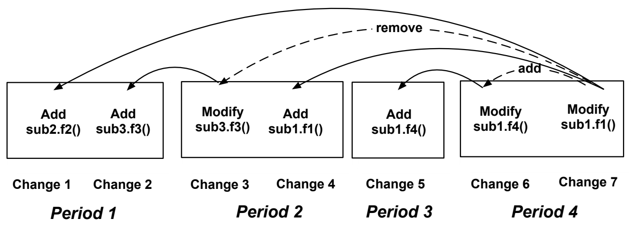
This section introduces the concepts used to measure the space, time and effort dimensions of knowledge preservation for subsystems of a software project. Similar to other work [6], a subsystem can be any logical (e.g., all functions collaborating on a major feature) or physical (e.g., file system directories) collection of source code files. Two of our measures, i.e., foundationality and sporadicity, are based on the concept of time dependence between source code entities. This concept was introduced in our previous work to track the progress of projects [10] and to detect foundational periods

```
void sub1.f1(void){
  sub2.f2();
  sub3.f3();
}
```

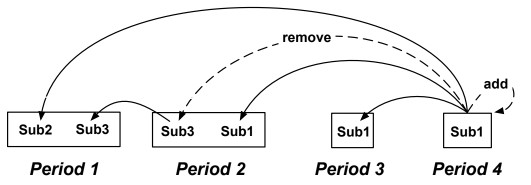
(a)

```
void sub1.f1(void){
  sub2.f2();
  sub1.f4();
}
```

(b)



(c)



(d)

Figure 1: 1a) and 1b show a source code snippet before and after source code change 7, respectively. The corresponding change-level time dependence relations are shown in Figure 1c. Figure 1d lifts up the change-level time dependence relations to the subsystem level.

of software systems [11]. Here, we refine the concept in space by allowing specific subsystems to be foundational at different points in time, instead of the whole system at once. The remainder of this section first outlines the necessary background information on time dependence, then presents all three measures used in this paper as well as how we implemented them.

2.1. Time Dependence between Entities

Entity-level time dependence establishes ***time dependence relations between different revisions of source code entities*** (functions and variables). An entity E that is changed at time T builds (depends) on the most recent revision of all entities to which accesses or calls existed (possibly removed now) or were added at time T. Figure 1 illustrates entity-level time dependence using a small example that runs across four time periods. Change 7 modifies function `f1()` in Figure 1a by removing the call to `f3()` and adding a call to `f4()`, resulting in Figure 1b. Hence, `f1()` in change 7 builds on the last revision of `f1()` itself (Change 4), the last revision of all entities called before Change 7 (`f2()` and `f3()`), and the last revision of newly called entities (`f4()`).

In this paper, we focus on time dependence in-the-large instead of on time dependence in-the-small. More in particular, we lift up time dependence from the level of individual changes/revisions to the changes' encompassing periods (such as quarters or years), and, more importantly, from time dependence between individual entities to time dependence between the *subsystems* to which those entities belong. Hence, time dependence has a notion of space (subsystems) and time (periods). Figure 1d shows how all change-level dependencies between entities in Figure 1 were lifted up to period-level entities between subsystems. Note that the "add" edge between "sub1.f1()" and "sub1.f4()" is lifted up into a self-edge. We can now define the concepts needed to study knowledge preservation.

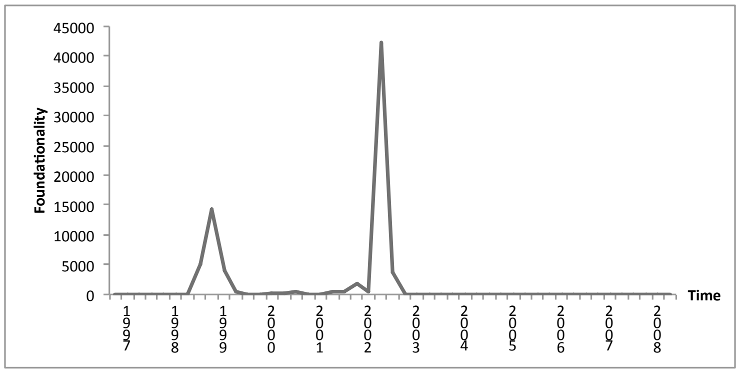
2.2. Measures for Knowledge Preservation

Space Dimension: Foundationality. ***Foundational subsystems and development periods*** are subsystems and periods that have a large impact on the development of other subsystems. The latter subsystems heavily access or call variables and functions developed or changed by a foundational subsystem in a foundational period. In other words, the changes to the foundational subsystem in a foundational development period triggered changes in many, possibly all dependent subsystems. As such, it is important to preserve knowledge of the specific development that happened during this period. If the subsystem's change in this development period would not have triggered changes in so many other subsystems, that development period would not be foundational for the subsystem and would seem less critical to preserve knowledge for.

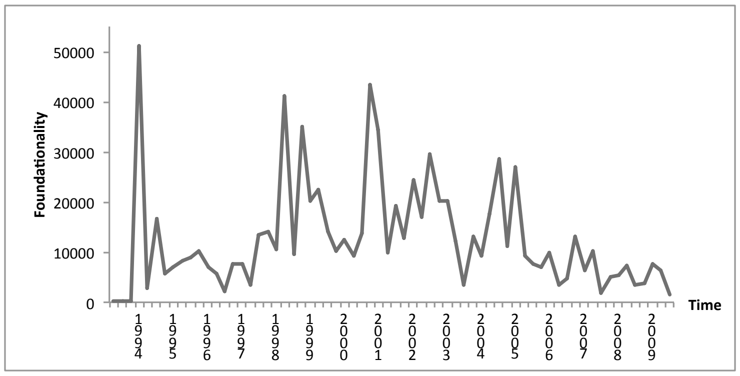
The ***foundationality of a subsystem S in a particular development period*** D formally is defined as the number of incoming time dependence relations for S in period D originating from subsystems in period D or later. A higher foundationality means that more subsystems changed later on because of the changes to S in D. For example, the foundationality of "Sub2" in "Period 1" is 1 (edge from "Sub1" in "Period 4"), because to make the changes to "Sub1" in "Period 4", one needs knowledge about "Sub2" in "Period 1". The foundationality of "Sub1" in "Period 4" is also 1 (self-edge).

The ***total foundationality of a subsystem*** is the sum of the subsystem's foundationalities across all development periods [12]. A higher total foundationality means that more subsystems changed later on because of changes to S. For example, "Sub1" has a total foundationality of 3, whereas for "Sub2" it is 1, meaning that "Sub1" has played a more foundational role in the lifetime of the project (its changes forced more dependent subsystems to change).

Time Dimension: Sporadicity. A second important concept is the ***sporadicity of a subsystem***. For example, Figure 2a plots the foundationality of PostgreSQL's **odbc** subsystem in each period (PostgreSQL is one of the case study subject systems). **odbc** is a "sporadically foundational" subsystem, since its foundationality is concentrated in only two, clearly distinct development periods. In contrast, FreeBSD's **kern** subsystem (FreeBSD is the second case study subject system) in Figure 2b experiences dramatic variation in foundationality, almost continuously throughout all development periods. Hence, **kern** can be considered a "continuously foundational" subsystem. For such subsystems, it can be very hard to prioritize development periods for knowledge



(a) PostgreSQL



(b) FreeBSD

Figure 2: The foundationality of the **odbc** subsystem in PostgreSQL and the **kern** subsystem in FreeBSD across development periods. **odbc** is more sporadically foundational than **kern**.

preservation, since dependent subsystems are impacted by changes to the foundational subsystem almost all the time.

We capture sporadicity of a subsystem in terms of the normalized entropy of foundationality over time [13], i.e.:

$$\begin{aligned} sporadicity(s) &= 1 - normalized\_entropy(s), \forall subsystems\ s \\ &= 1 + \frac{1}{log_2\left(n\right)} \times \sum_{i=1}^{n} p_i(s) \times log_2\left(p_i(s)\right), \forall subsystems\ s \end{aligned}$$

with

$$\begin{aligned} p_i(s) &= \frac{foundationality\ of\ s\ in\ period\ i}{total\ foundationality\ of\ s} \\ \sum_{i=1}^{n} p_i(s) &= 1 \end{aligned}$$

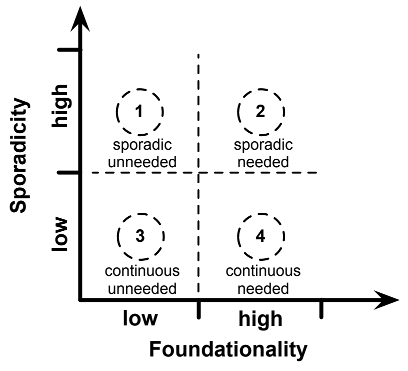


Figure 3: Comparison of the concepts of foundationality and sporadicity.

$$n \quad = \quad total\ number\ of\ development\ periods$$

This entropy expresses how uniform the distribution of foundationality is across time. A normalized entropy of 1 means that foundationality is uniformly distributed across time (“continuous”), whereas a normalized entropy of 0 means that all foundationality is concentrated in one development period (“sporadic”). Since we do not want to measure uniformity, but sporadicity, we subtract the normalized entropy from 1. In other words, if each development period experiences the same foundationality, the sporadicity is 0 ($p(s) = \frac{1}{n}$), i.e., the foundationality is continuous. If, on the other hand, all foundationality is focused in one development period, then sporadicity is 1 ($p_j(s) = 1$ and $p_i(s) = 0, \forall i \neq j$). The sporadicity of **odbc** is 0.63, whereas the sporadicity of **kern** is 0.07, i.e., the foundationality of **odbc** is much more sporadic. This is because **odbc** was migrated to a different source control repository in 2002 [11], causing the extremely low foundationality afterwards (Figure 2a).

Effort Dimension: Number of Changes. The effort involved with preserving the knowledge of a subsystem in a particular development period follows from the amount of new knowledge about a subsystem added in that period. To approximate this amount of new knowledge, we use the number of source code changes to the subsystem in that period. Alternative measures could be the size of source code changes or the volume of messages on mailing lists, but a 100% accurate measure is hard to achieve.

Discussion. The relation between foundationality and sporadicity is illustrated in Figure 3. Subsystems in zones 1 and 3 are not that foundational (we call them “hardly foundational”), and hence have a lower need (priority) for up-to-date preservation of project knowledge. If one would be interested in the subsystems in zone 3, typically knowledge would need to be preserved for most of the subsystems’ development periods. Zones 2 and 4 theoretically correspond to the most foundational subsystems (“highly foundational”). Preservation of knowledge is recommended for both zones, but for subsystems in zone 2 it is easier to prioritize knowledge preservation, since there are only a limited number of development periods to consider.

The number of changes and foundationality are two different measures, since it is possible for development periods to be foundational based on only one source code change, for example a new version of a library that is imported into a source code repository. All dependent subsystems will need to update their dependency on this library, introducing a substantial number of time dependency edges, and hence increasing foundationality.

The opposite is also possible, i.e., a non-foundational development period with thousands of source code changes, for example to the "main" function of a system. Although our definition of foundationality takes these changes into account via time dependence self-edges, the absence of any incoming relation from other subsystems generally leads to a low foundationality. Hence, such changes typically end up with a low priority for knowledge preservation, unless other dependencies would be taken into account in addition to time dependence. We consider this to be future work.

2.3. Implementation

The source control repository of a project (e.g., CVS or SVN) contains all the information required to calculate foundationality, sporadicity and the number of changes. The main issue is that such repositories typically contain rather low-level information. Instead of subsystem-level information like "subsystem 1 now depends on subsystem 2, which was last changed 1 quarter ago" or at least source code entity-level information like "a function call to g was added to function f, and g was last changed 5 weeks ago", repositories typically contain line-level information like "line 5 was changed on the 8th of December 2011".

In order to lift up the line-level information to the subsystem- and period-level required for our purposes, one can use evolutionary extractors like C-REX [14]. Such extractors statically analyze all change transactions that happened over time. They parse the source code changes to identify added and removed function calls and variable accesses, then link these calls and accesses to the files containing the corresponding function and variable definitions. Since static analysis is used, this linking is not 100% accurate. For example, two files could both contain a function with the same name. Ideally, the actual build configuration should be considered to know which of the two files is really used in a particular release of the product, however we did not do this for this paper (nor for our previous work [10, 11]).

In this paper, we use the C-REX evolutionary extractor and some scripts that lift up function- and week-level information to subsystem- and quarter-level, and calculate the metrics discussed in this section. C-REX ignores changes done for indentation or copyright updates [15]. Furthermore, it follows a lexical approach to process source code changes, not a full parser. This allows processing uncompilable code changes, but (as mentioned above) slightly reduces the accuracy of the linking. To address this, we use heuristics based on the most specific common super-folder. For example, if files "/a/b/c/d.c" and "/a/b/e/f.c" contain a function named "f" and a third file "/a/b/c/g.c" calls "f", the most specific common super-folder of "/a/b/c/d.c" and "/a/b/c/g.c" is "a/b/c", whereas for "/a/b/e/f.c" and "/a/b/c/g.c" it is "a/b". Since "a/b/c" is longer than “a/b”, we resolve the call to “f” to “/a/b/c/d.c”, since the latter file likely is related more closely to “/a/b/c/g.c”.

3. Research Questions

Using our methodology, we study three research questions, one for each dimension of knowledge preservation:

Q1 Which subsystems should have a higher priority for knowledge preservation?

According to the previous section, foundational subsystems are the subsystems that we would give a high priority for knowledge preservation. Hence, we are interested in which subsystems are foundational for a given software project. Do foundational subsystems provide core functionality (i.e., system libraries or components with crucial APIs that provide the essential structure for other subsystems), or can non-core end user subsystems be foundational as well?

Q2 Which development periods should have a higher priority for knowledge preservation?

Does a subsystem typically exhibit short bursts of foundationality (sporadically foundational subsystem), or is it uniformly foundational throughout the lifetime of the project (continuously foundational subsystem)? It is easier for sporadically foundational subsystems to determine the development periods that should have the highest priority for knowledge preservation.

Q3 How much effort is involved in preserving knowledge of foundational subsystems?

In previous work [11], we found that foundational periods typically experienced a high number of source code changes. If the same holds at the level of foundational subsystems, preserving knowledge for foundational subsystems will require substantial effort. Hence, in this question we study if there is a correlation between the foundationality of subsystems in development periods and the number of changes performed to these subsystems.

4. Case Study

To explore the three research questions, we performed a case study on two large, long-lived open source projects. We first present the two studied systems, then present the results for our three questions.

Studied Systems

For our case study, we used the source code histories of the open source PostgreSQL (1996–2008) and FreeBSD (1993–2009) projects, as explained by Table 1. PostgreSQL is a relational database system of which the original design goes back to the 1980s [18], whereas FreeBSD is an operating system distribution derived from the

Table 1: Characteristics of the studied systems.

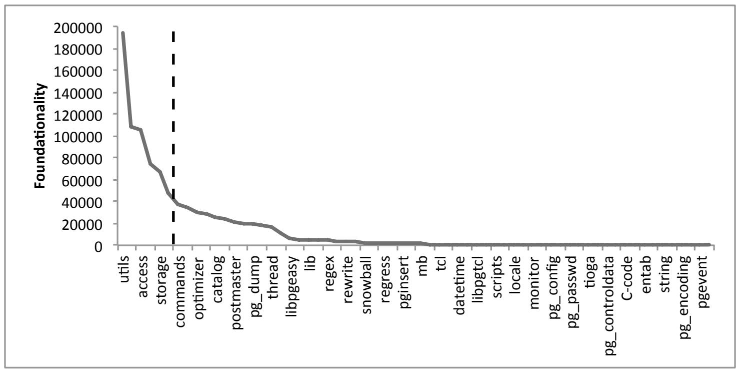
	PostgreSQL	FreeBSD
type	DBMS	Operating System
CVS module	pgsql/ [16]	src/ [17]
period	10/1996–06/2008	07/1993–12/2009
#quarters	47	66
#changes	84,311	1,074,858
#entities	31,863	617,000
#files	2,053	37,724
#bug fixes	22,913	144,582
#subsystems	64	957

Berkeley flavour of UNIX [19]. We studied the FreeBSD system including the kernel. We used quarters (3 sequential months) as "period", since it is a common time period for project planning (other time periods could easily be explored using our approach) [20]. We picked both systems due to their long and archived history of changes (Table 1), and our experience with them from our prior work [10, 11]. The two systems being from two different domains (databases and operating systems) helps us validate the generality of our findings across different domains.

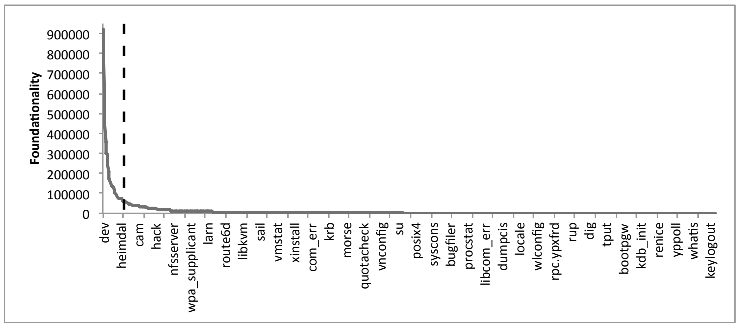
Before starting our case studies, we studied the available documentation for PostgreSQL and FreeBSD. On the one hand, since both projects have academic roots and later gathered a large developer and user community, many books, papers and tutorials have been written on PostgreSQL and FreeBSD. On the other hand, keeping this documentation up-to-date requires substantial effort.

For example, both projects dedicate specific developers to documentation and use collaborative media like wikis to actively involve users in the documentation process ("Consider contributing your knowledge back." [21]). Furthermore, FreeBSD has a dedicated "documentation project" with explicit todo lists [22]. The FreeBSD bug report system also lists numerous entries for outdated documentation [23]. At the time of writing (November 2010), there were 37 critical documentation bug reports, and 293 non-critical ones. For example, there was no documentation for "The New SCSI layer for FreeBSD (CAM)", and large parts of the architecture handbook and USB audio support were outdated. PostgreSQL has a smaller list (6 entries) of open problem reports related to documentation [24], which mainly contains more technical issues such as migration to other documentation formats.

To address our research questions, we used the approach outlined in C-REX (Figure 2.3) on the CVS repositories of PostgreSQL and FreeBSD (Table 1). As proposed by prior work [6], this paper considers the second level directories as subsystems of PostgreSQL (e.g., **odbc** in /interfaces/odbc/Attic/) and the fourth level directories as subsystems of FreeBSD (e.g., **dev** in /freebsd/src/sys/dev/cxgb/). As mentioned earlier, we wrote scripts to calculate the three metrics from Section 2.2.



(a)



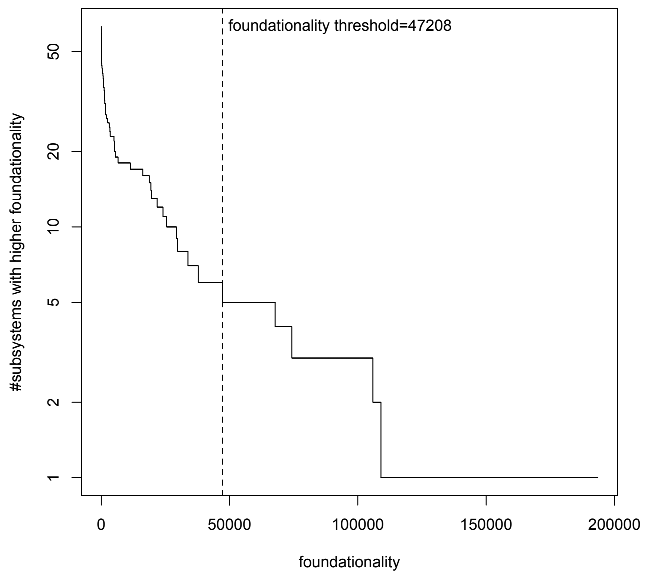
(b)

Figure 4: Distribution of total foundationality across the subsystems of (a) PostgreSQL and (b) FreeBSD. The dashed line represents the border between highly and hardly foundational subsystems based on our threshold.

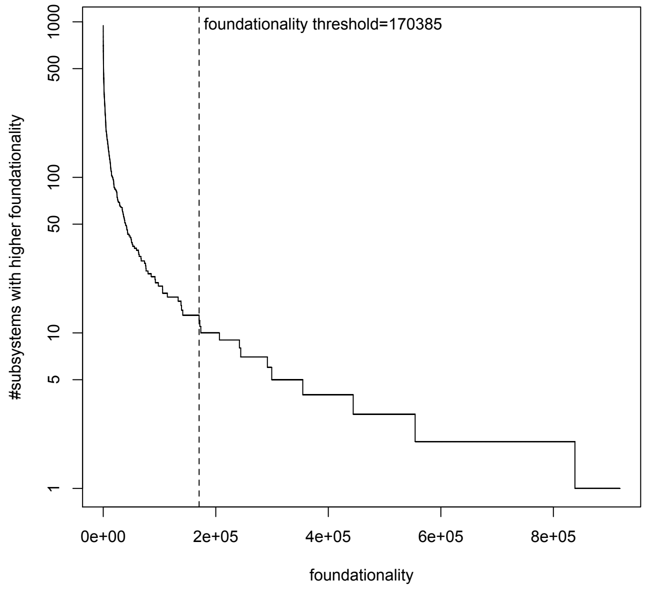
Q1. Which subsystems should have a higher priority for knowledge preservation?

To facilitate our discussion, we picked a meaningful threshold to distinguish between hardly and highly foundational subsystems. By no means this is the only possible threshold, since the right threshold to use depends on the resources (such as time and personnel) available to an organization for preserving knowledge. If more resources are available, a lower threshold should be used to consider more subsystems as highly foundational. Figure 5 plots the distribution of the number of highly foundational subsystems for the range of foundationality values of the subsystems in PostgreSQL and FreeBSD. We can see that this number changes slowly for high foundationality values, yet for small values the number increases rapidly. This suggests diminishing returns when picking a threshold that is too low.

To focus our discussion, we selected a threshold based on the differences (deltas) in foundationality between neighbouring subsystems, as visualized by the horizontal lines in Figure 5. These differences are roughly decreasing towards less foundational subsystems. Hence, we selected as highly foundational all subsystems starting from the subsystem with the highest foundationality down to the first subsystem with a delta

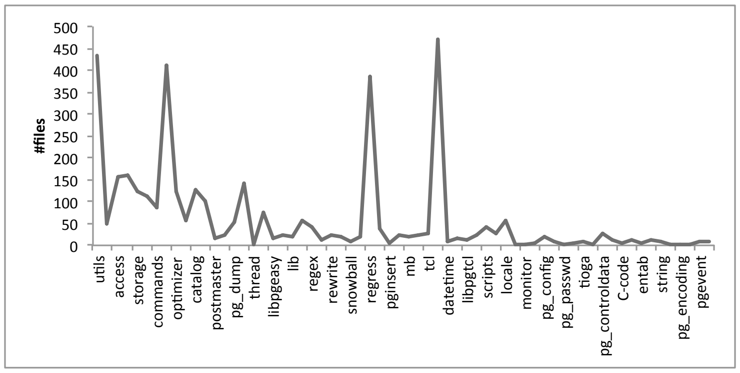


(a)

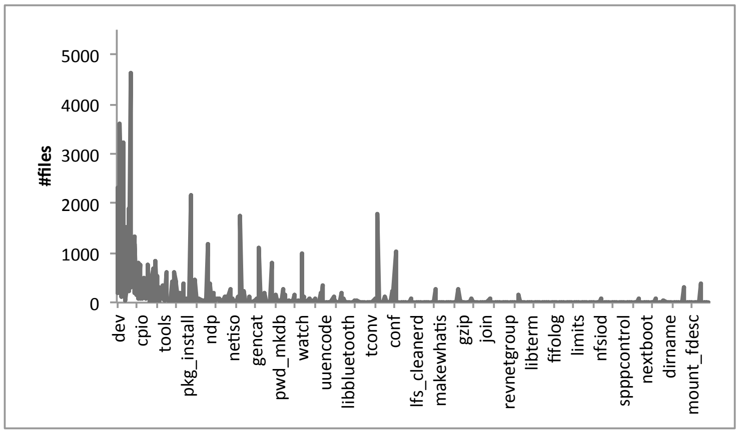


(b)

Figure 5: Plot with for all the possible choices of foundationality threshold in (a) PostgreSQL and (b) FreeBSD the number of subsystems with higher foundationality, i.e., the number of highly foundational subsystems. The dashed line shows the threshold that we used. The delta in foundationality of a subsystem corresponds to the length of the horizontal line ending at the subsystem's foundationality value.



(a)



(b)

Figure 6: Distribution of the #files in each subsystem in the last studied period of (a) PostgreSQL and (b) FreeBSD. The subsystems are sorted by decreasing total foundationality (cf. Figure 4). Subsystems that no longer exist in the last studied period were awarded the median #files across all subsystems.

larger than a particular value (one tenth of the maximum delta, i.e., 8,452 for PostgreSQL and 28,398 for FreeBSD). This threshold corresponds to the dashed vertical line on Figures 4a, 4b and 5, which separates hardly (left) and highly (right) foundational subsystems.

Figures 4a and 4b plot the distribution of total foundationality across the subsystems of PostgreSQL and FreeBSD. We observe that only a small percentage of subsystems have a high total foundationality. Table 2 lists the top 20 most foundational subsystems of PostgreSQL and FreeBSD, including all highly foundational ones (based on our threshold). Only 9.4% of the subsystems in PostgreSQL (6 out of 64) and 1.4% of the subsystems in FreeBSD (13 out of 957) are highly foundational.

The highly foundational subsystems all provide core functionalities. For example, the top 5 subsystems in PostgreSQL are **utils** (built-in data types and routines for memory management, database transactions and text encoding), **nodes** (structure for stor-

Table 2: Table showing for the top 20 most foundational subsystems for PostgreSQL (out of 64) and for FreeBSD (out of 957): (1) total foundationality (Found.), (2) sporadicity (Spor.), and (3) the Spearman correlation between each quarter's foundationality and total number of changes. Bold numbers for Found. highlight highly foundational subsystems, whereas bold numbers for Spor. highlight sporadically foundational subsystems.

PostgreSQL				FreeBSD			
Subsystem	Found.	Spor.	Corr.	Subsystem	Found.	Spor.	Corr.
utils	**193,533**	0.11	0.68	dev	**918,375**	0.05	0.79
nodes	**109,014**	0.37	0.86	kern	**838,289**	0.07	0.67
access	**105,867**	0.15	0.61	i386	**554,306**	0.18	0.94
odbc	**74,289**	**0.63**	0.99	sys	**444,248**	0.19	0.40
storage	**67,762**	0.18	0.75	gcc	**354,610**	**0.61**	0.98
libpq	**47,208**	0.14	0.63	user.bin	**299,445**	**0.49**	0.92
commands	37,798	0.05	0.79	libc	**291,723**	0.23	0.78
port	33,798	**0.51**	0.82	netinet	**244,350**	0.12	0.67
optimizer	29,798	0.16	0.91	boot	**242,008**	0.30	0.91
parser	29,269	0.18	0.91	net	**206,489**	0.18	0.68
catalog	25,506	0.16	0.87	gdb	**173,414**	**0.73**	0.98
executor	24,073	0.17	0.82	contrib	**171,629**	0.35	0.95
postmaster	21,782	0.15	0.79	binutils	**170,385**	**0.68**	1.00
tcop	19,597	0.20	0.75	pc98	141,303	**0.45**	0.97
pg_dump	19,357	0.21	0.85	vm	139,107	0.20	0.90
ecpg	18,708	0.30	0.91	amd64	138,220	0.15	0.84
thread	16,210	**0.80**	1.00	perl5	133,159	**0.96**	1.00
psql	11,314	0.27	0.96	libstdc++	113,946	**0.76**	1.00
libpgeasy	6,564	**0.99**	0.88	openssh	105,528	**0.49**	1.00
initdb	5,461	**0.55**	1.00	openssl	105,429	**0.68**	1.00

ing SQL queries), **access** (query algorithms based on b-trees and r-trees), **odbc** (API for accessing PostgreSQL on the Windows platform [25]), and **storage** (manages the PostgreSQL storage system). In FreeBSD, the top 5 subsystems are **kern** (kernel implementation), **dev** (device drivers), **i386** (architecture-specific kernel implementation for the i386 platform), **sys** (kernel header files), and **gcc** (GCC compiler).

Hardly foundational subsystems turn out to be subsystems that either do not provide essential functionality, or represent "consumer" subsystems like end user applications and scripting engines. A consumer subsystem only builds on (consumes) other subsystems, without providing functionality to other subsystems in return. Such subsystems are less important to preserve knowledge for, since they are not of interest to developers of other subsystems.

Examples of hardly foundational subsystems in PostgreSQL are **soundex** (user-defined function for matching based on similar sounding names), **pg_encoding** (utility to check encoding of data) and **pg_id** (id utility for shell scripts), whereas examples of consumer subsystems are **cli** (command line interface), **main** (main module of PostgreSQL) and **python** (Python interface). Examples of hardly foundational subsystems in FreeBSD are **scrshot** (screenshot utility), **setpmac** (run command with different MAC process label) and **fib** (Fibonacci heap library), whereas examples of consumer subsystems are **perl** (Perl interpreter), **dnsquery** (DNS query utility) and **keylogin** (decryption tool).

Are foundational subsystems by definition larger than non-foundational ones? Figures 6a and 6b plot the number of files for each subsystem (in the last studied period), ordered by decreasing total foundationality (cf. Figure 4). For subsystems that did not exist anymore in the last studied period, we used the median file size across all subsystems.

Highly foundational subsystems tend to be larger than hardly foundational ones. In PostgreSQL, the median number of files for highly foundational subsystems is 139.5 files compared to 19 for hardly foundational subsystems, whereas in FreeBSD it is 1,469 compared to 6. Overall, the Spearman correlation between foundationality and file size is 0.73 for PostgreSQL and 0.69 for FreeBSD, i.e., moderately high. Hence, hardly foundational subsystems tend to be larger.

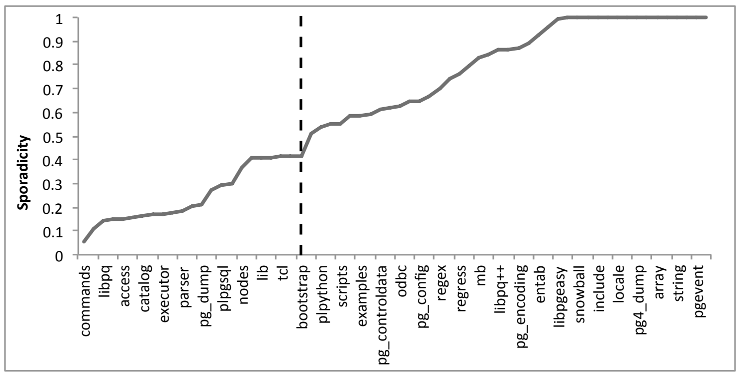
Of the four largest subsystems of PostgreSQL, **utils** is highly foundational, whereas (from left to right) **port** (substitute system APIs for the Windows platform), **regress** (regression test and infrastructure) and **include** (shared interfaces across all subsystems) are hardly foundational. For FreeBSD, **user.bin** (UNIX system utilities), **libc** (C standard library) and **dev** are highly foundational. **openssl** (SSL/TLS library) on the other hand is hardly foundational (#20 in Table 2).

Highly foundational subsystems, i.e., subsystems of which knowledge should be preserved first, correspond to large, core subsystems. Subsystems with lower priority to preserve knowledge for either provide less essential functionality, or represent consumer subsystems like end user applications.

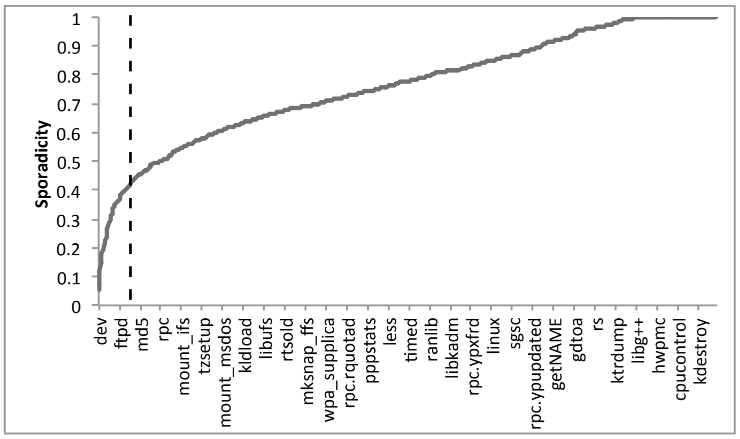
Q2. Which development periods should have a higher priority for knowledge preservation?

In the previous research question, we prioritized foundational subsystems from a knowledge preservation point of view. However, do such subsystems exhibit sporadic periods of foundationality, or are they continuously foundational throughout the lifetime of a project? A subsystem that exhibits only sporadic periods of foundationality intuitively should be easier to preserve knowledge for, since only a relatively limited number of development periods is foundational. On the other hand, subsystems that are foundational throughout the lifetime of a project continuously undergo restructuring and refactoring that impact hundreds of other subsystems. Such continuously foundational subsystems require knowledge preservation of most (if not all) development periods, since there is no single most foundational development period with highest priority.

Figures 7a and 7b plot the distribution of sporadicity across subsystems, sorted in descending order. These curves are clearly different from each other. Whereas the distribution of sporadicity follows a concave trend for FreeBSD, PostgreSQL follows a much more accidental trend, with some plateaus.



(a)

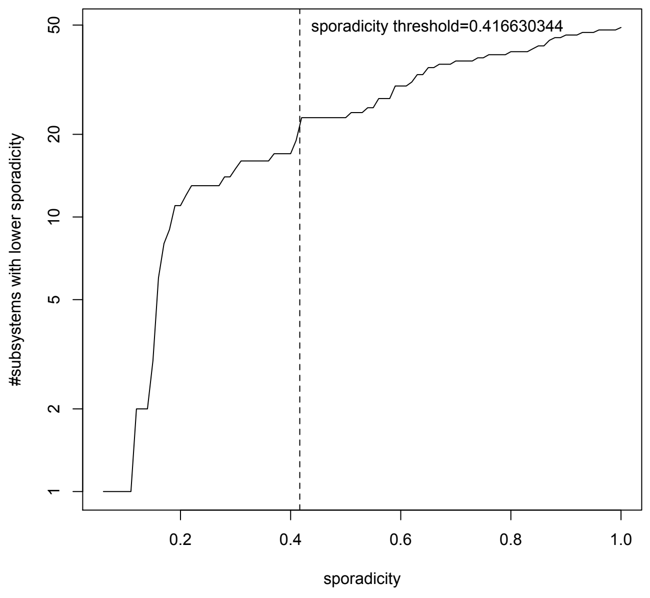


(b)

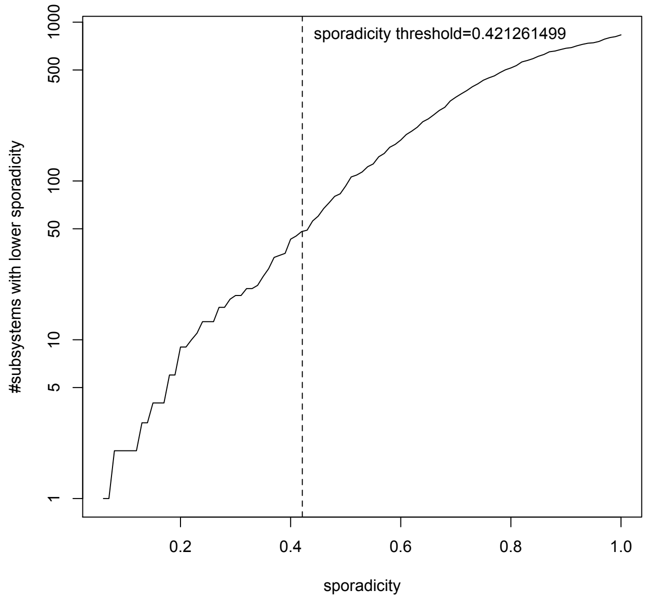
Figure 7: Distribution of sporadicity across (a) PostgreSQL and (b) FreeBSD subsystems. The dashed line represents the border between sporadically and continuously foundational subsystems.

Similar to foundationality, the right sporadicity threshold to use depends on the number and kind of resources available. To determine a sporadicity threshold for our discussion, we basically use the same delta-based methodology as for the foundationality threshold in Q1. This time, Figure 5 plots the number of subsystems with *lower* sporadicity for the range of sporadicity values of the subsystems in PostgreSQL and FreeBSD (sorted from low to high). Since the deltas in sporadicity in Figure 8 (length of horizontal lines) do not follow the steady downward trend of for example Figure 5, we manually picked the threshold between both groups by looking for an out-of-place long delta (this is the most clear for PostgreSQL). With the dashed line thresholds, 35.9% of the PostgreSQL subsystems (23 out of 64) and 5.1% of the FreeBSD subsystems (49 out of 957) would be continuously foundational.

Using these thresholds, we now analyze sporadically and continuously foundational subsystems in PostgreSQL and FreeBSD, then compare them to highly and hardly

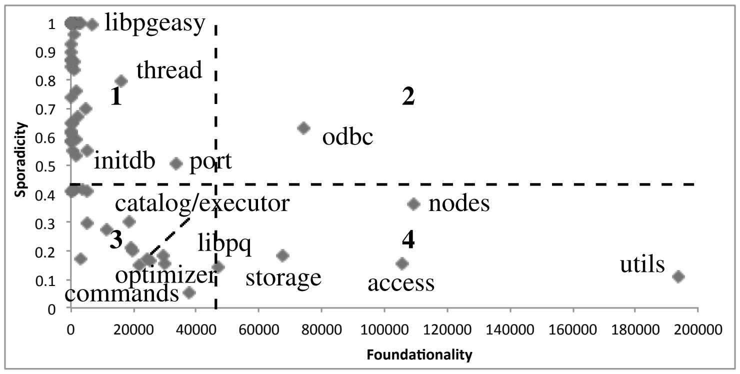


(a)

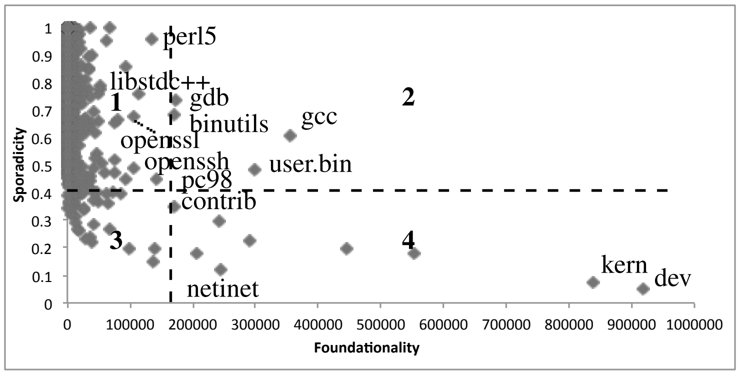


(b)

Figure 8: Plot with for all the possible choices of sporadicity threshold in (a) PostgreSQL and (b) FreeBSD the number of subsystems with lower sporadicity, i.e., the number of sporadically foundational subsystems. The dashed line shows the threshold that we used. The delta in sporadicity of a subsystem corresponds to the length of the horizontal line ending at the subsystem's sporadicity value.



(a)



(b)

Figure 9: Scatterplot of sporadicity vs. foundationality for (a) PostgreSQL and (b) FreeBSD.

foundational subsystems.

Sporadically Foundational Subsystems in PostgreSQL

The most continuously foundational subsystems of PostgreSQL are **commands** (random collection of portal and utility support code), **utils** (see question Q1) and **libpq** (PostgreSQL front-end library). Other core components of PostgreSQL, such as **postmaster** (dispatcher of front-end queries to back-end), **optimizer** (query plan generation) and **catalog** (system catalog manipulation) are also highly continuously foundational.

The most sporadically foundational PostgreSQL subsystems are those subsystems with the lowest foundationality. Many of these subsystems were introduced in the very first quarter of the PostgreSQL project, but were never foundational again afterwards (no changes triggered other subsystems to change). One exception is **libpgeasy** (simplified version of **libpq**), which is the 19th most foundational subsystem in Table 2. Four other highly foundational subsystems are sporadically foundational as well, i.e.,

thread (threading API), **odbc** (see question Q1), **initdb** (database initialization) and **port** (see question Q1).

Sporadically Foundational Subsystems in FreeBSD

The most continuously foundational subsystems of FreeBSD are **dev** (see question Q1), **kern** (kernel implementation) and **netinet** (IP/TCP protocol stack). Similar to PostgreSQL, most of the continuously foundational subsystems are core subsystems. The findings for sporadically foundational subsystems are also similar to those of PostgreSQL. There are nine sporadically foundational subsystems among the 20 most foundational subsystems in Table 2, i.e., **perl5** (Perl), **libstdc++** (C++ runtime support), **gdb** (debugger), **binutils** (linking and assembly tools), **openssl** (see question Q1), **gcc** (compiler), **user.bin** (see question Q1), **openssh** (secure network protocol) and **pc98** (port of FreeBSD for the NEC PC-98x1 architecture).

All of these subsystems are core development tools or libraries that are not maintained by the FreeBSD development team. Instead, they are imported at very specific moments in time for customization [11], which can trigger significant changes in other subsystems. This explains why these subsystems are sporadically foundational. Of course, this does not mean that the imported subsystems do not evolve in between the import times. The full development history is maintained in the subsystems' own source code repository where all regular development occurs. FreeBSD only sees the major snapshots.

Sporadicity vs. Foundationality

Up until now, we have found that the top foundational subsystems generally tend to be continuously foundational, which is relatively bad news from the point of knowledge preservation (hard to prioritize a specific period to preserve knowledge for). Here, we are interested in better understanding the specific relation between foundationality and sporadicity. For this, we use a scatter plot visualization similar to Figure 3.

Figures 9a and 9b provide interesting insights into the differences between PostgreSQL and FreeBSD. We demarcated the four zones of Figure 3 based on the thresholds used for foundationality in question Q1 and sporadicity earlier in this section. Although other thresholds obviously will shift subsystems around between the four zones, the major trends discussed below remain similar.

In PostgreSQL, most subsystems are in zone 1, with some subsystems in zones 3 and 4, and even fewer in zone 2. This means that most subsystems do not urgently require knowledge preservation (zones 1 and 3). Of those subsystems that should have a higher priority for knowledge preservation (zones 2 and 4), most are continuously foundational (zone 4), i.e., most of their development periods should get a high priority for knowledge preservation. **odbc**, **port** and **thread** are the sporadically foundational subsystems with highest foundationality.

In FreeBSD, the situation is different. The majority of subsystems belongs to zones 1 and 3, followed by zones 4 and 2. The two most foundational subsystems, i.e., **dev** and **kern**, clearly require knowledge preservation in almost every development period. The highly foundational subsystems that are sporadically foundational (i.e., **gdb**, **binutils**, **gcc** and **user.bin**) are mostly developed upstream (**gdb**, **binutils**

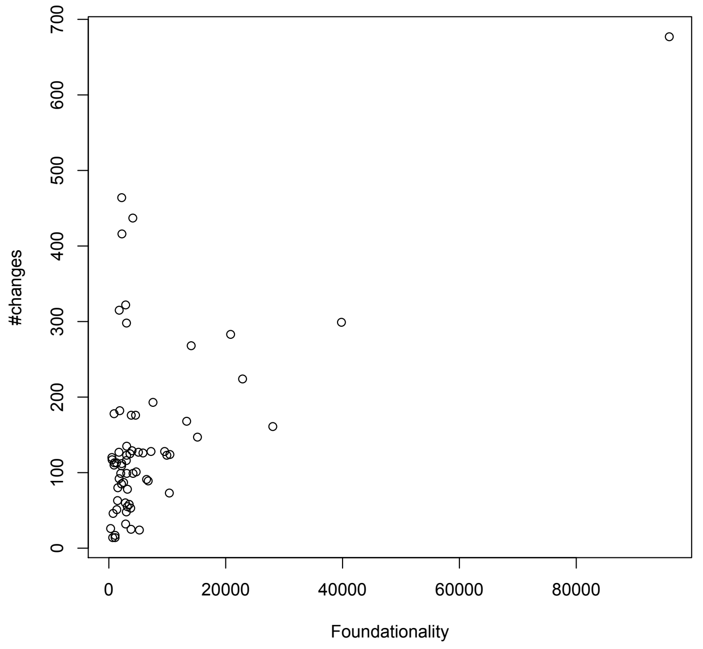


Figure 10: Scatterplot of the #changes versus the foundationality of the development periods of the **sys** subsystem of FreeBSD.

and **gcc**) or are already mature subsystems (**user.bin**). In the former case, each period in which an upstream project is imported potentially could have a high impact on other subsystems and hence warrant knowledge preservation to answer questions like "which APIs were changed?" and "how to update to the new API version?".

Most highly foundational subsystems are continuously foundational throughout the lifetime of a project, whereas most hardly foundational subsystems are sporadically foundational.

Q3. How much effort is involved in preserving knowledge of foundational subsystems?

To study the effort involved with preserving knowledge for subsystems, we use the number of changes to a subsystem in a period as a proxy. We calculate two types of Spearman rank correlations (we use a non-parametric correlation, since the data is not normally distributed):

Global correlation We calculate the Spearman rank correlation between the total number of changes and the total foundationality of each *subsystem*, to get a rough indication of whether foundationality and number of changes correlate across all subsystems.

Subsystem-level correlation Per subsystem S, we calculate the Spearman rank correlation between the number of changes to S and the foundationality of S, across all *quarters*. This tells us for individual subsystems whether or not development periods of subsystems for which knowledge should be preserved are typically associated with substantial effort.

We find that the global correlation is very high: 0.87 for PostgreSQL and 0.91 for FreeBSD. At the subsystem-level (see Table 2), the correlation is very high for most of the subsystems. This suggests that highly foundational subsystems are not only more continuously foundational, but also that preserving the knowledge of foundational development periods and keeping the knowledge up-to-date needs to consider a significantly larger amount of changes (i.e., effort).

These observations are confirmed when analyzing the median total number of changes for highly foundational subsystems and for hardly foundational subsystems. Highly foundational subsystems have a significantly higher median number, i.e., 4,451.5 vs. 87 for PostgreSQL and 22,575 vs. 69 for FreeBSD.

There is only one highly foundational subsystem that has a low subsystem-level correlation. **sys** (see question Q1) has a correlation of only 0.40. This follows from the fact that some *less* foundational quarters saw *more* changes, as shown in Figure 10. Since **sys** consists of kernel header files, it is indeed obvious that small changes might impact a large number of subsystems. The top right data point corresponds to the oldest development period that we considered.

Most foundational subsystems are not only continuously foundational, they also experience substantially more code changes, hence requiring more substantial effort to preserve knowledge and keep this knowledge up-to-date.

5. Threats to Validity

Threats to construct validity [26] relate to whether our measurements quantify what we intended them to. Our calculation of time dependence is primarily based on quarters. Other periods like months, years, releases could be used and might lead to different findings. Also, our time dependence relations are derived from static call graph dependencies. Implicit dynamic dependencies are not captured. Therefore, we miss some of the time dependence relations between changes.

Since our technique is based on the historical source code changes archived in source control repositories, the most recent development periods and subsystems typically will have lower foundationality than older periods. This does not mean that it is not important to preserve knowledge for the former periods and subsystems. We are currently experimenting with a weighting system to eliminate the skew towards older development periods and subsystems.

We define subsystems as a fixed level in the file system hierarchy of PostgreSQL (second level) and FreeBSD (fourth level), as explained in Section 4. Although this approach is well-known (e.g., [6]), and provides us with subsystems for which concrete project documentation is available, other definitions of subsystem should be explored. For example, subsystems might be defined at different levels in the future: a subsystem S might consist of directories C and Y in */A/B/C/* and */X/Y/Z/*.

We used the number of changes in a development period as a proxy for the amount of knowledge that should be preserved, i.e., the effort involved with preservation. This is a simplification, since a small change like switching the implementation of for example "malloc" requires careful documentation of the rationale of the switch and the implementation details of the new version. Similarly, renaming a method that is called throughout the system is a trivial change that does not require major preservation.

Our analysis is based on two thresholds, i.e., one for foundationality and one for sporadicity. Although we documented our methodology for determining the thresholds, this methodology by no means is the only acceptable one. Different thresholds can be used according to the needs and resources of the practitioner. If significant people and funding are available, lower thresholds can be used (more highly foundational and continuously foundational systems). Otherwise, higher thresholds should be used (less highly foundational and continuously foundational systems). In our case studies, we showed how the number of highly foundational and sporadically foundational subsystems evolves for all possible threshold values.

External validity relates to the generalization of our study results. Our case studies are based on two open source C systems. Although they come from different domains (database and operating system), our results may not generalize to systems of other domains or commercial systems. Also, systems in different programming languages or even paradigms (OO) might show different results. Additional studies are needed to investigate this.

Finally, our approach focuses on supporting practitioners in prioritizing subsystems for which knowledge should be preserved. This approach was used in this paper to retro-actively study which development periods of which subsystems are most important to preserve knowledge for. However, to prove the effectiveness of our approach in pro-actively guiding practitioners, we need to perform a user study.

6. Related Work

In this section, we discuss the closest related work on the different dimensions of knowledge preservation. We also relate this paper to our earlier work on time dependence. Brudaru and Zeller propose to measure the genealogy of changes, which uses a directed acyclic graph to model the impact of changes on defects [27]. They build this genealogy by iteratively establishing change dependencies at the level of lines of code. During this process, the changes that break the system are observed and the impact of changes on future defects is analyzed. Although their approach was not validated in practice, Brudaru et al.'s concept of change dependency has some similarity to our approach. Our approach considers information from the source control repositories at subsystem-level and across time instead of at line-level.

German et al. [28] introduce the concept of Change Impact Graph (CIG) to detect the impact of a change on its dependent changes when changing a source code entity. They iteratively visualize the call graph of a function and call graphs of its called entities within a time window. The approach aims primarily at locating bugs of a function. Our approach aims to assist practitioners to identify subsystems and development periods for which knowledge preservation is needed.

Existing software evolution research does not explore the temporal dependence between changes. Usually, software metrics such as LOC [29, 30] are measured to monitor and detect the development periods with rapid or slow growth. Tilley et al. [5] reverse-engineer the subsystem dependencies in one snapshot of a software system for re-documentation purposes. They do not consider the time and effort dimension.

The work on change coupling between software entities, like classes and methods, analyzes what other parts of the code need to be changed if a given piece of code is changed [31]. However, the dependencies studied in that line of work relate entities to other entities in the same version of the code base, whereas time dependence relates a change of an entity to past changes of itself and other called entities.

Kothari et al. [32] introduce the concept of canonical changes to categorize the change clusters of a project into different areas or activities, like maintenance and new development. Our approach is somewhat similar in intent, but focuses on the effort to preserve and maintain knowledge.

Other time-related research analyzes historical data to better understand large, long-lived software systems. Mockus et al. [33] use historical data from version systems to identify code experts. Chen et al. [34] developed a tool called CVSSearch, which uses the CVS comments to track source code fragments. Hassan and Holt [35] introduce the idea of attaching *Source Sticky Notes* to static dependency graphs, which assist in better understanding the software architecture. Our approach leverages time dependence between changes to identify the foundational subsystems. Kim et al. [36] trace the evolution of software clones in a clone group across time, i.e., they identify time dependence of clones.

There is quite some effort-related work on bug prediction techniques. Most recent techniques are based on process metrics derived from the change history of a system [37, 38, 4, 39]. These techniques typically look at code churn [39] or the number of changes to a file. Bernstein et al. [37], for example, use the number of revisions and reported issues in the last quarter to predict the location and number of bugs in the next month. More recently, different techniques have been proposed to factor bug fixing effort into bug prediction models [40]. In future work, we plan to compare the performance of our approach against approaches that use other types of historical data.

Finally, in our previous work [10], we used the concept of time dependence to assist project managers to track the progress of their project. We also studied the impact of the most recent time dependence on the appearance of bugs. To study the foundational periods of a software project [11], we considered time dependence relations at the entity level instead of at the subsystem level. This paper lifts up time dependence between source code entities to subsystems and studies the characteristics of these subsystems over their evolution, i.e., we consider both space and time.

7. Conclusion

To keep track of the software development process and to support future evolution of a software system, preserving and maintaining related software artifacts and their metadata is indispensable. However, at the same time such knowledge preservation is hard to achieve because of the continuous evolution of software projects in size and complexity, whereas only a limited amount of project resources are available for preservation. This paper proposes an automated technique to analyze which subsystems to prioritize for knowledge preservation without having to spend too much effort. Our approach is based on the foundationality and sporadicity metrics that can be derived from the time dependence relations between the subsystems of a project, and on the number of source code changes stored in the source control repository. Basically, highly foundational subsystems are subsystems whose changes in a particular development period typically trigger a massive amount of changes in dependent subsystems.

Through a case study on two large open source systems, we find that, as could be expected, highly foundational subsystems mostly correspond to relatively large, core subsystems. These subsystems definitely should get a high priority when preserving and updating knowledge. However, most of those subsystems are continuously foundational, i.e., almost all development periods are important to preserve knowledge for. In addition, the high number of changes in those periods means that knowledge preservation requires substantial effort.

Although our technique can support practitioners to automatically recommend valuable subsystems for which knowledge preservation is feasible with reasonable effort, there is definitely room for future work. In particular, what foundationality and sporadicity thresholds should be used in what context? How should one prioritize the subsystems in a specific zone in the foundationality-sporadicity scatter plot? How should one deal with the large group of continuously foundational subsystems for which knowledge preservation has a high priority (zone 4)? Who should preserve the knowledge of a particular subsystem, and who has the required knowledge? Finally, does missing to preserve important knowledge really lead to significantly more bugs and wasted development effort?

Acknowledgments. The authors want to thank Weiyi Shang and the anonymous reviewers for their insightful suggestions and comments.

References

[1] U. Dekel, J. D. Herbsleb, Improving API documentation usability with knowledge pushing, in: Proc. of the 31st Intl. Conf. on Software Engineering (ICSE), Vancouver, BC, Canada, 2009, pp. 320–330.

[2] T. Fritz, G. C. Murphy, Using information fragments to answer the questions developers ask, in: Proc. of the 32nd ACM/IEEE Intl. Conf. on Software Engineering - Volume 1 (ICSE), Cape Town, South Africa, 2010, pp. 175–184.

[3] T. D. LaToza, G. Venolia, R. DeLine, Maintaining mental models: a study of developer work habits, in: Proc. of the 28th Intl. Conf. on Software Engineering (ICSE), Shanghai, China, 2006, pp. 492–501.

[4] M. Leszak, D. E. Perry, D. Stoll, Classification and evaluation of defects in a project retrospective, J. Syst. Softw. 61 (2002) 173–187.

[5] S. R. Tilley, H. A. Müller, M. A. Orgun, Documenting software systems with views, in: Proc. of the 10th annual international conf. on Systems documentation (SIGDOC), Ottawa, ON, Canada, 1992, pp. 211–219.

[6] M. W. Godfrey, Q. Tu, Evolution in open source software: A case study, in: Proc. of the Intl. Conf. on Software Maintenance (ICSM), San Jose, CA, US, 2000, pp. 131–142.

[7] S. Koch, Software evolution in open source projects—a large-scale investigation, Journal of Software Maintenance and Evolution 19 (6) (2007) 361–382.

[8] G. von Krogh, S. Spaeth, K. R. Lakhani, Community, joining, and specialization in open source software innovation: a case study, Research Policy 32 (7) (2003) 1217 – 1241.

[9] Y. Wang, D. Guo, H. Shi, Measuring the evolution of open source software systems with their communities, SIGSOFT Software Engineering Notes 32 (6) (2007) 7–13.

[10] O. Alam, B. Adams, A. E. Hassan, Measuring the progress of projects using the time dependence of code changes, in: Proc. of the 25th IEEE Intl. Conf. on Software Maintenance (ICSM), Edmonton, AB, Canada, 2009, pp. 329–338.

[11] O. Alam, B. Adams, A. E. Hassan, A study of the time dependence of code changes, in: Proc. of the 16th Working Conf. on Reverse Engineering (WCRE), Lille, France, 2009, pp. 21–30.

[12] R. C. Holt, Structural manipulations of software architecture using tarski relational algebra, in: Proc. of the Working Conf. on Reverse Engineering (WCRE), Honolulu, HI, US, 1998, pp. 210–219.

[13] C. E. Shannon, Prediction and entropy of printed english, Bell System Technical Journal 3 (1951) 53–64.

[14] A. E. Hassan, Mining software repositories to assist developers and support managers, Ph.D. thesis, University of Waterloo, Waterloo, ON, Canada (2004).

[15] A. E. Hassan, Automated classification of change messages in open source projects, in: Proc. of the 2008 ACM Symposium on Applied Computing (SAC), Fortaleza, Ceara, Brazil, 2008, pp. 837–841.

[16] PostgreSQL, Cvs repository (pgsql/ module), `:pserver:anoncvs@postgresql.org:/usr/local/cvsroot`.

[17] FreeBSD, Cvs repository (src/ module), `anoncvs@anoncvs1.FreeBSD.org:/home/ncvs`.

[18] `http://www.postgresql.org/`.

[19] `http://www.freebsd.org/`.

[20] A. E. Hassan, R. C. Holt, The chaos of software development, in: Proc. of the 6th Intl. Wrksh. on Principles of Software Evolution (IWPSE), Helsinki, Finland, 2003, pp. 84–94.

[21] `http://developer.postgresql.org/pgdocs/postgres/resources.html`.

[22] `http://www.freebsd.org/docproj/todo.html`.

[23] `http://www.FreeBSD.org/cgi/query-pr-summary.cgi?category=docs&responsible=`.

[24] `http://wiki.postgresql.org/wiki/Todo#Source_Code`.

[25] `http://www.postgresql.org/developer/ext.backend_dirs.html`.

[26] R. K. Yin, Case Study Research: Design and Methods - Third Edition, SAGE Publications, London, 2002.

[27] I. I. Brudaru, A. Zeller, What is the long-term impact of changes?, in: Proc. of the intl. wrksh. on Recommendation Systems for Software Engineering (RSSE), Atlanta, Georgia, 2008, pp. 30–32.

[28] D. M. German, A. E. Hassan, G. Robles, Change impact graphs: Determining the impact of prior codechanges, Inf. Softw. Technol. 51 (2009) 1394–1408.

[29] H. Gall, M. Jazayeri, J. Krajewski, CVS release history data for detecting logical couplings, in: Proc. of the 6th Intl. Wrksh. on Principles of Software Evolution (IWPSE), Helsinki, Finland, 2003, pp. 13–23.

[30] M. M. Lehman, J. F. Ramil, P. D. Wernick, D. E. Perry, W. M. Turski, Metrics and laws of software evolution - the nineties view, in: Proc. of the 4th Intl. Symposium on Software Metrics (METRICS), Albuquerque, NM, US, 1997, pp. 20–32.

[31] S. Mirarab, A. Hassouna, L. Tahvildari, Using bayesian belief networks to predict change propagation in software systems, in: Proc. of the 15th IEEE Intl. Conf. on Program Comprehension (ICPC), Banff, AB, Canada, 2007, pp. 177–188.

[32] J. Kothari, A. Shokoufandeh, S. Mancoridis, A. E. Hassan, Studying the evolution of software systems using change clusters, in: Proc. of the 14th IEEE Intl. Conf. on Program Comprehension (ICPC), Athens, Greece, 2006, pp. 46–55.

[33] A. Mockus, L. G. Votta, Identifying reasons for software changes using historic databases, in: Proc. of the Intl. Conf. on Software Maintenance (ICSM), San Jose, CA, US, 2000, pp. 120–130.

[34] A. Chen, E. Chou, J. Wong, A. Y. Yao, Q. Zhang, S. Zhang, A. Michail, CVSSearch: Searching through source code using CVS comments, Proc. of the IEEE Intl. Conf. on Software Maintenance (ICSM) 0 (2001) 364–373.

[35] A. E. Hassan, R. C. Holt, Using development history sticky notes to understand software architecture, in: Proc. of the 12th IEEE Intl. Wrksh. on Program Comprehension (IWPC), Bari, Italy, 2004, pp. 183–192.

[36] M. Kim, V. Sazawal, D. Notkin, G. Murphy, An empirical study of code clone genealogies, in: Proc. of the 10th European software engineering conf. held jointly with the 13th ACM SIGSOFT intl. symp. on Foundations of software engineering (ESEC/FSE-13), Lisbon, Portugal, 2005, pp. 187–196.

[37] A. Bernstein, J. Ekanayake, M. Pinzger, Improving defect prediction using temporal features and non linear models, in: 9th intl. wrksh. on Principles of software evolution (IWPSE), Dubrovnik, Croatia, 2007, pp. 11–18.

[38] T. L. Graves, A. F. Karr, J. S. Marron, H. Siy, Predicting fault incidence using software change history, IEEE Transactions on Software Engineering 26 (7) (2000) 653–661.

[39] N. Nagappan, T. Ball, Use of relative code churn measures to predict system defect density, in: Proc. of the 27th Intl. Conf. on Software engineering (ICSE), St. Louis, MO, US, 2005, pp. 284–292.

[40] T. Mende, R. Koschke, M. Leszak, Evaluating defect prediction models for a large evolving software system, in: Proc. of the European Conf. on Software Maintenance and Reengineering (CSMR), Madrid, Spain, 2009, pp. 247–250.