A Study of the Time Dependence of Code Changes

Omar Alam, Bram Adams and Ahmed E. Hassan
Software Analysis and Intelligence Lab (SAIL)
School of Computing, Queen's University, Canada
{omar, bram, ahmed}@cs.queensu.ca

***Abstract*—Much of modern software development consists of building on older changes. Older periods provide the structure (e.g., functions and data types) on which changes in future periods will build. Given a particular period in the lifetime of a project, one can determine prior periods on which it builds, and future periods which build on it. Using this knowledge, managers can identify foundational periods in the lifetime of a project, which provide the structural foundation for a large number of future periods. A good understanding and detailed documentation of events and decisions in such foundational periods is essential for the smooth evolution of a project. This paper examines how changes build on older changes by measuring the time dependence between code changes. Using our approach, we can create time dependence relations between periods and study the characteristics of such dependence relations. We apply our approach on two large open source projects, PostgreSQL and FreeBSD. We find that foundational periods are periods with huge restructurings, important new features or large imports of external source code. We also find that a project, as it ages, either progressively depends on older periods or cycles between depending on old and new periods.**

***Keywords*-D.2 Software Engineering: Maintenance management, Maintenance measurement, Maintenance planning, Restructuring, reverse engineering, and reengineering, Review and evaluation, Software Architectures, Project control and modeling. K.6 Management of Computing and Information Systems: Software Management Software maintenance.**

I. Introduction

New code changes typically build on prior changes. For example, a new function would depend on (i.e., call or use) other code entities that are added in the same change, or have previously been defined. The previously defined entities are either system libraries or were added in prior changes. A temporal dependence exists between a new code change and prior changes. This temporal dependence flows as well between periods in a lifetime of a project. For example, changes (e.g., refactorings and new APIs) done during a particular year might pave the way for changes in future periods (years).

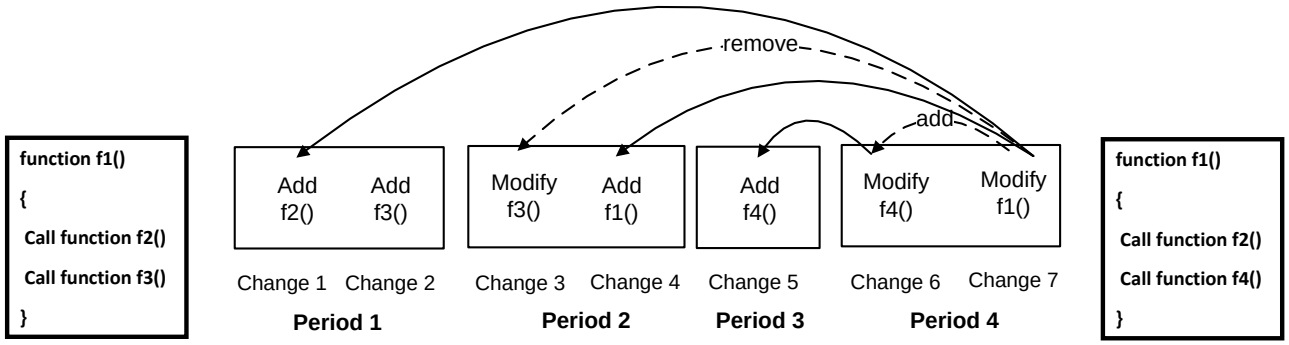
Such temporal dependence between periods has not been studied before, although this dependence has a strong impact on the smooth and successful evolution of a project. For example, it allows to identify foundational periods, i.e., periods of development on which a significant number of changes in later development build. Managers should schedule extra testing and validation effort when source code from a foundational period is changed. They may also decide to re-document the development activities of foundational periods to ensure that resources like documentation and mailing lists are archived and up-to-date. Since the foundational periods have crucial impact on later developments, staff during those periods should be consulted, if needed, for better understanding and smooth development of future periods.

A good understanding of temporal dependence would help software evolution researchers in understanding the state of a project. As a project stabilizes, we would expect the number of foundational periods to drop. Indeed, most changes for such projects would be small and incremental changes which build on previously established structure provided by past foundational periods. While actively evolving, we would expect the appearance of new foundational periods which provide the structure for future changes.

This paper presents an approach to measure the time dependence between code changes and between development periods. Using our approach, we study the time dependence between code changes and periods, for two large open source systems (PostgreSQL and FreeBSD). We find that:

- On average, up to 48% of the changes in a period build on changes made in the current development period.
- As a project ages, it tends to build less on changes from the current period, and more on foundational, older periods.
- Foundational development periods (on which a large number of future periods build) are periods with huge restructurings, important new features or large imports of external source code.

Organization of the Paper. The paper is organized as follows. Section II presents our methodology to measure the time dependence between changes and periods. Section III presents three research questions we want to study using our approach. In Section IV, we explain the setup of the two case studies we performed, and we discuss the case study results for each research question. Section V discusses limitations and future work. Section VI discusses related work. Section VII summarizes our findings and concludes the paper.



(a) function before changes

(b) changes introduced to entities over time

(c) function after changes

Figure 1. Time dependence relations for an example system before (1a) and after (1c) making changes. The arrows in 1b connect changes 6 and 7 to all changes they build on. The dashed arrow connects a change to the most recent change of any entity to which it added or removed a reference (e.g. function call).

II. Methodology

Software evolution studies usually track the progress of a project using traditional metrics like Lines of Code (LOC) between different versions of a software system [1], [2]. In this paper we wish to track the evolution of a project along the analogy of a building structure with new changes building on prior changes and new periods building on older periods.

We consider a ***change*** as the insertion, deletion or modification of a source code entity. Related changes are committed together into a change list to the source code repository of a project. To track the evolution of changes, we define a ***time dependence*** relation ***between changes***. A time dependence relation links a change of a source code entity E (e.g., a function or type definition) at time T to the most recent change before T of E and of each entity that E depends on before or after the change (via its call graph). The latter makes sure there is a time dependence relation corresponding to added *and* removed function calls.

A ***time dependence relation between periods*** (e.g., month, quarter, year) abstracts the time dependencies between changes to the time periods in which these changes reside. A time dependence between periods gives a higher-level view of the time dependence relations in terms of periods instead of individual changes.

Figure 1 illustrates a small example of the time dependence between changes and periods. The example has changes happening across four time periods. In change 7, a developer modifies function `f1()` (Figure 1a) by removing the call to `f3()` and adding a call to `f4()` (Figure 1c). Such a change builds on the last change to the function `f1()` itself, and on the last change to all previously (`f2()` and `f3()`) and newly (`f4()`) called entities. The time dependence edges express these dependencies. Rectangular nodes represent periods and edges represent time dependence relations between changes in these periods. Figure 1b only shows edges for changes 6 and 7, but similar edges can be drawn for other changes as well. The time dependence relation between two periods corresponds to the union of edges between any two changes in these two periods. For example, between periods 3 and 4, there is only one time dependence relation (small black edge).

We define ***inner- and outer-period time dependencies***. Inner-period dependencies are when an edge starts and ends in the same period, i.e., self-loops. These indicate that a period builds on itself. Outer-period dependencies are when an edge starts in a period and ends in a different period.

We can also measure the ***backward and forward time dependence of a period***. The backward time dependence of a period produces the set of all outgoing time dependence edges. These edges point to all prior periods on which this current period builds. The larger the number of edges produced when calculating the backward time dependency, the more this period builds on prior periods. The forward time dependence of a period produces the set of all time dependence edges pointing to that period. The source nodes of these edges correspond to all future periods which build on this current period. The larger the number of edges produced when calculating the forward time dependency, the more foundational a period is. Periods on which a large number of other periods build are considered to be ***foundational periods***.

We can also study the ***age of a time dependence relation*** , which is the time difference between the source and destination periods of a time dependence relation. For example, the age of the backward time dependence relation between change 4 and change 7 in Figure 1b is two periods.

We expect that in practice systems have ***foundational periods*** on which new changes continuously build. These periods contribute essential code which forms the structural foundation on which future changes build. An example of such structural foundation are changes which define APIs

or platform libraries on which other code changes build. Also, the first import of code into the source code repository provides a structural foundation on which many changes build. As a project evolves, we expect that new foundational periods will emerge whenever there are major restructurings and rework.

The required information for calculating the time dependence edges is readily available in the source control repository of a project. Our approach uses information at the entity level while most source control systems store change data at the line level. We developed a technique which lifts the line-level change information to the level of code entity changes, like functions, function calls and variables [3]. Using this entity-level information, we can reconstruct the call graphs for functions and build the time dependence relation for each source code change. By decorating time dependence information with metadata attached to each change [4], such as the change commit message and the name of the developer responsible for the change, we can figure out the prior code changes on which a change builds. We developed a lexical approach which helps us detect and ignore changes done for indentation or copyright update purposes [5].

III. Research Questions

Using the calculated time dependence relations between periods, we sought to explore the following three research questions. We briefly state and motivate the relevance of each question:

Q1 How does the distribution of time dependence on vary over time?

Do projects in general build on old periods, more recent periods or inner-period dependencies? In the latter two cases, a project builds on recent features and changes, whereas if it builds on much older periods, the project is likely in maintenance mode. Based on the results of this question, we can get a better understanding of the evolution of software projects using a building structure analogy instead of using metrics such as LOC.

Q2 As projects age, do they build more on older periods?

Do projects continuously build on older periods when they age or does their dependence on old and new periods vary? The former hints at a mature project, whereas the latter gives indications about a cyclic development process.

Q3 What are the foundational periods in a lifetime of a project?

Are there periods on which changes happening months or even years later still build? Identifying such periods permits us to focus on improving our knowledge of such periods. Managers should ensure that developers with crucial experience about these periods are retained. Moreover, managers should archive and enhance any missing documentation from these periods. Examples of such documentation can be change requests, requirements and important email or mailing list discussions.

	PostgreSQL	FreeBSD
type	DBMS	Operating System
period	1996–2007	1993–2005
#changes	84,311	353,958
#entities	31,863	253,896
#files	2,053	21,093

Table I
Characteristics of the studied systems.

IV. 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 we present the results for our three research questions.

A. Studied Systems

For our case studies, we used data from the open source PostgreSQL (1996–2007) and FreeBSD (1993–2005) projects. PostgreSQL is a relational database system of which the original design goes back to the 1980s [6], whereas FreeBSD is an operating system distribution derived from the Berkeley flavor of UNIX [7]. We studied the FreeBSD base system, which corresponds to the kernel and a limited number of crucial system utilities like run-time libraries and a compiler. We studied the time dependence at the level of a quarter, since it is a common time period for project planning [8] (other time periods could be explored using our approach). We picked both systems due to their long and archived history of changes, as Table I shows. The two systems being from two different domains (databases and operating systems) would help us verify the generality of our findings across domains.

Q1. How does the distribution of time dependence on vary over time?

In general, projects building on old periods face problems when recruiting new developers, as new people need to learn about these older, most likely undocumented periods [9]. On the other hand, building on changes made in the current period might be risky as the code is still fresh and relatively untested compared to the older code, leading to the appearance of more bugs [10]. Hence, it is important to understand how the time dependence on older periods varies over time.

To study the variance in the time dependence of periods, we calculate the backward time dependencies of each period in the lifetime of a project. Then, we measure the age of

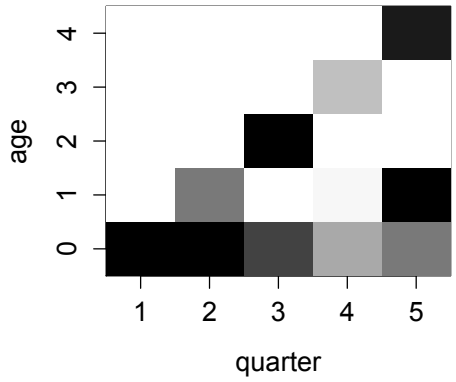


Figure 2. Illustration of a heatmap showing the distribution of backward time dependencies over time. Darker cells mean that a period has more backward time dependencies of that age. Age is calculated in quarters.

each of these backward time dependencies for each quarter. To facilitate our analysis, we graphically rendered this information as a heatmap. Figure 2 is a simple illustration of a heatmap to explain the concept, whereas Figure 3a and Figure 3b show the actual heatmaps for PostgreSQL and FreeBSD, respectively.

A colored cell with coordinate (3,2) in Figure 2 means that quarter 3 builds on changes from two quarters ago, i.e. these changes are two quarters old. An age of zero corresponds to building on changes from the current quarter, but we do not show this in Figure 3a and Figure 3b (more on this later). The darker the color of cell (3,2), the more changes from two quarters ago quarter 3 builds on. The color is relative to the whole heatmap, i.e. black corresponds to the highest number of backward dependencies across all quarters and ages, whereas white means that there are no backward time dependencies to that period. As the coloring is relative to the whole heatmap, black cells in Figure 3a and Figure 3b represent different numbers of backward time dependencies. Hence, we cannot compare the absolute colors between the PostgreSQL and FreeBSD heatmaps, but we can compare the relative coloring patterns within a particular heatmap to study the distribution of backward time dependencies over time.

Each column of cells in Figure 2 shows how the age of the backward time dependencies of a quarter is distributed over time. As shown on Figure 3a and Figure 3b, this distribution is not uniform, and it varies widely across quarters. We briefly summarize our most pertinent findings.

On average, up to 48% of the backward time dependencies are inner-period dependencies.

Figure 4a and Figure 4b show that the percentage of inner-period backward time dependencies of a period varies between 18.4% and 100% for PostgreSQL and 21.3% and 100% for FreeBSD. The averages are 40.4% and 47.9% respectively. The most foundational period for each quarter is the quarter itself. We do not include dependencies of age 0 (i.e., inner-dependencies) in our heatmap, otherwise they would dominate the plot with a very dark line across all periods and with other cells being much lighter in color.

Quarters with many changes do not always build on more changes

Some columns in the heatmaps are very lightly colored compared to the other columns, like for example quarter 4 in Figure 2, whereas others are very dark. This is because the corresponding quarters respectively are less or more dependent on previous periods. A peak or a low in the total amount of changes on which each quarter depends corresponds to a relatively darker/lighter column in the heatmaps.

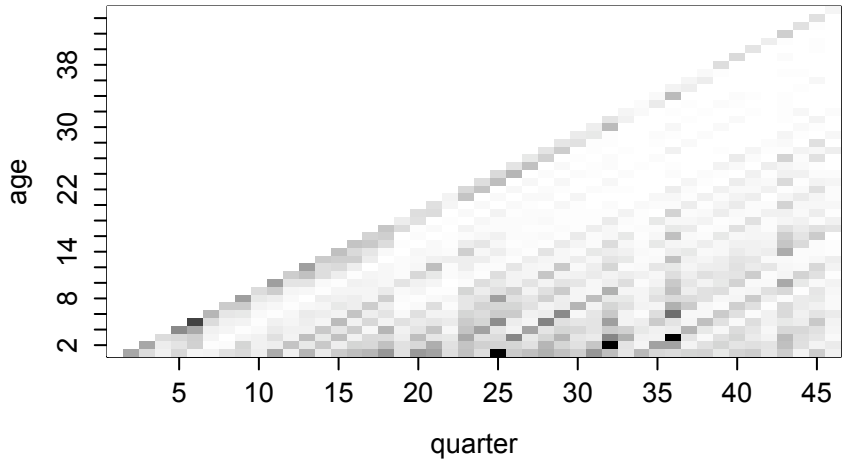
It seems intuitive that a quarter would build more on prior quarters (dark column) if more source code changes have been performed in that quarter. The Pearson correlation between the total amount of backward time dependencies of a quarter and the total number of changes made in that quarter is 0.51 for PostgreSQL and 0.87 for FreeBSD. Hence, in FreeBSD higher developer activity in a period indeed means that the changes in that period depend more on older periods as they have more backward time dependencies. However, in PostgreSQL there are multiple periods in which this conjecture does not hold, with a large number of changes being done, yet these changes not building on older periods.

On average, up to 48% of the changes in a period build on changes done in the same period. Quarters with many changes do not always indicate a dependence on older periods.

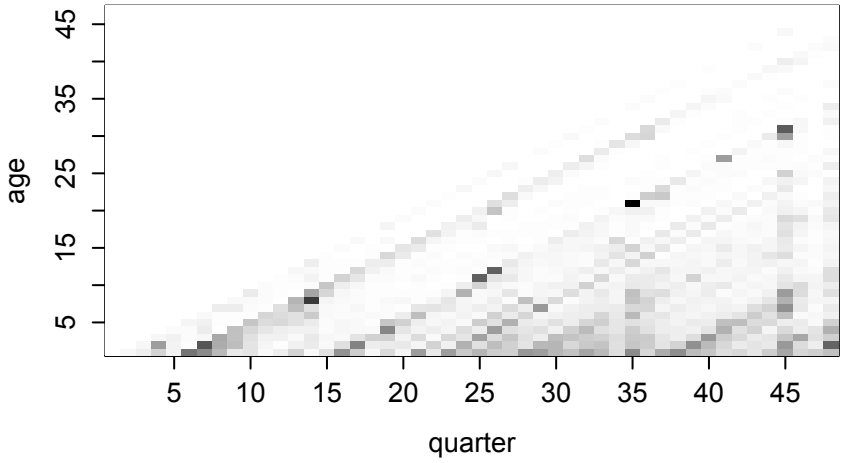
Q2. As projects age, do they build more on older periods?

Intuitively, we expect that projects become more stable over time. Applied to the topic of this paper, this would mean that projects would build more on older changes as the projects age. Periods in which the age of backward time dependencies suddenly decreases give strong indications of huge restructurings or the addition of new features on which later periods tend to build. To study the evolution of backward time dependencies over time, we use the boxplots in Figure 5a and Figure 5b, in addition to the heatmaps.

The boxplots show for each quarter how the age of backward time dependencies is distributed over time. The extreme whiskers on the boxplots correspond to the minimum and maximum age. The minimum age is always zero (which represents inner-period dependencies), whereas every period still builds on the first period of development (maximum age). The boxes in the boxplots show the lower quantile (bottom of a box), median (line in the middle of a box) and upper quantile (top of a box). These respectively mean that 25/50/75% of the backward time dependencies of a quarter is younger than the value of the lower quantile/median/upper



(a) PostgreSQL



(b) FreeBSD

Figure 3. Heatmaps showing the distribution of backward time dependencies through the lifetime of both projects. For each period (i.e., column), the cells vary in darkness based on the number of backward time dependencies of that age relative to all dependencies for all quarters. Darker cells indicate more dependencies at that age. Age is calculated in quarters.

quantile. The distance between the lower and upper quantile is called the "inter-quartile range" (IQR). We briefly explain our findings below.

PostgreSQL progressively builds on older periods

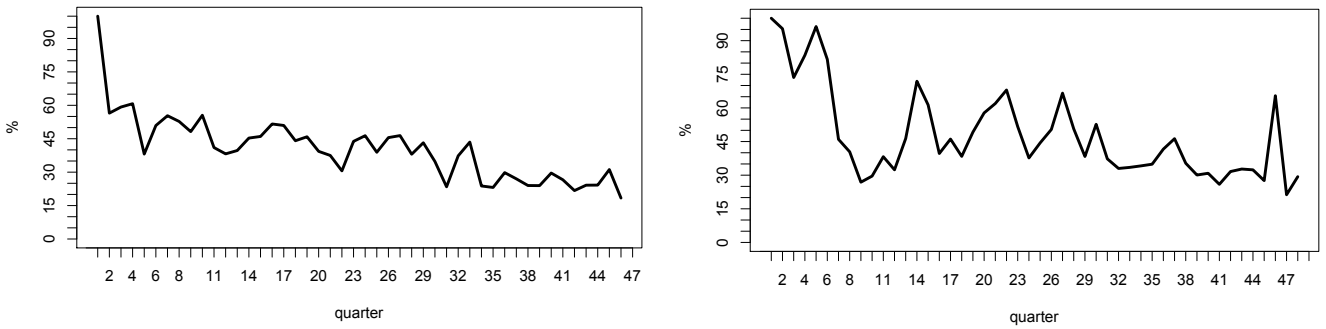
Backward time dependencies in PostgreSQL are slightly more widespread over time (larger IQR) than those in FreeBSD. In other words, PostgreSQL progressively builds on old changes. This becomes more obvious when looking at the median age of the backward time dependencies for PostgreSQL. The black curve formed by the medians of adjacent boxplots forms an almost continuous curve that shows two trends. Until quarter 34, the curve roughly followsa constant trend. From quarter 34 on (see Figure 5a), the curve steadily increases into a steep trend. In addition, the bottoms of the boxplots shift up to age three or even four, and the IQR grows slightly. Hence, PostgreSQL builds more on older changes as it ages.

This means that by quarter 34 PostgreSQL has reached such a degree of stability that most changes can just build on a proven foundation instead of requiring invasive changes that would lead to new foundational periods.

FreeBSD periodically cycles between old and recent periods

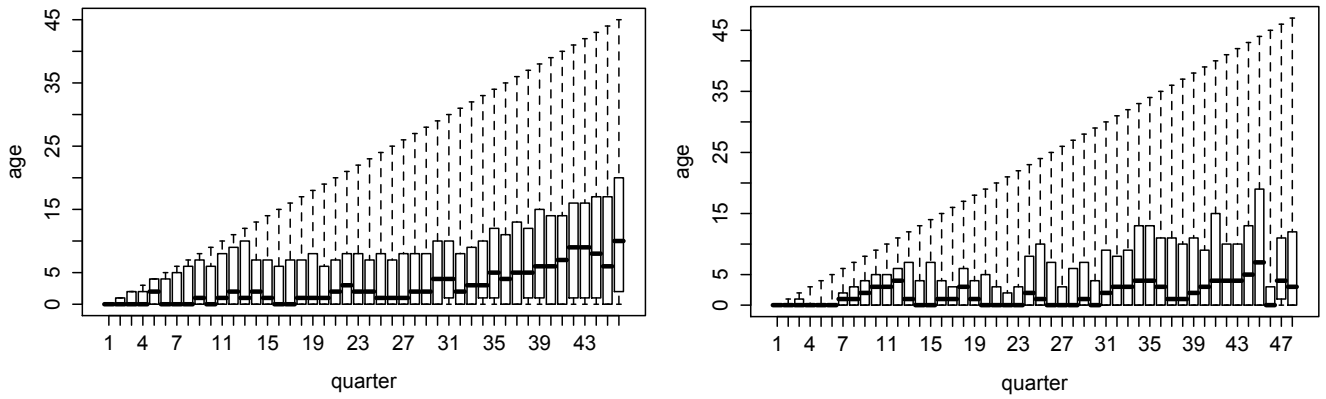
For FreeBSD, the median curve shows a very strong periodical trend starting from quarter 7 as shown in Figure 5b. We see periods building on earlier recent periods (low median) followed by an aging median, which suddenly resets to zero again. Between quarters 30 and 45, the median strongly increases (similar to PostgreSQL), but at quarters 37 and 46 the median age of backward time dependencies resets back to zero.

Comparing Figure 5b with Figure 4b, we find that the median age of backward time dependencies jumps back to



(a) PostgreSQL (b) FreeBSD

Figure 4. The percentage of inner-period backward time dependencies of a quarter.



(a) PostgreSQL (b) FreeBSD

Figure 5. Boxplots showing the minimum, lower quartile, median, upper quartile and maximum of the age of backward time dependencies of a quarter. Age is calculated in quarters.

zero when there are peaks in the relative percentage of inner-period backward dependencies. These peaks actually coincide with peaks in the total amount of changes on which a quarter is built. We manually investigated these observations using the FreeBSD commit logs. The peaks corresponding to quarters like 14, 27, 37 and 46 are due to the importing of new versions of large, externally developed source code systems like GCC, binutils, openssh and sendmail to the FreeBSD repository. These systems are imported because the FreeBSD base system combines the FreeBSD kernel with crucial externally developed system tools like compilers and libraries. These external tools are imported and significantly customized to better integrate them with the FreeBSD core. The effect of these modifications gradually fades out, after which the median age of backward time dependencies increases again. Hence, the periodical evolution of the age of backward time dependencies as time goes by is inherent to the nature of FreeBSD.

FreeBSD builds more on recent periods than PostgreSQL

The heatmap of FreeBSD in Figure 3b shows that backward time dependencies are concentrated on recent periods, as the color of old changes in the columns quickly fades out. PostgreSQL has a more even distribution of backward time dependencies over time, i.e. overall the cells are darker. Figure 4a and Figure 4b corroborate this: the percentage of inner-period backward dependencies for FreeBSD is frequently up to 20% higher than for PostgreSQL.

It took 1.5 year before FreeBSD started building on older periods

The periodical aging and renewing of backward time dependencies in FreeBSD only starts from quarter 7. Before, FreeBSD almost completely built on inner-period changes, as the boxes of boxplots disappear before quarter 7 in Figure 5b unlike PostgreSQL. Similarly, darker colors for the FreeBSD heatmap (Figure 3b) appear later in FreeBSD's evolution (at quarter 7) than for PostgreSQL (Figure 3a).

This pattern means that FreeBSD underwent significant overhauls during its first one and a half year before it got more stable and later quarters could start to build on the established foundation. This seems strange, as FreeBSD was a fork of the very stable Berkeley BSD operating system line. An investigation of the history of FreeBSD reveals

that the initial FreeBSD releases (December 1993) were based on the 4.3BSD-Lite ("Net/2") operating system from Berkeley [7]. Then there was a lawsuit between Novell and Berkeley after which the 4.3BSD-Lite operating system was deemed contaminated. FreeBSD had to be rewritten completely based on incomplete fragments of another operating system (4.4BSD-Lite). It took until the end of 1994, i.e. quarter 7, before FreeBSD was stable again.

PostgreSQL progressively builds on older changes, whereas FreeBSD shows a periodical trend in the age of backward time dependencies.

Q3. What are the foundational periods in a lifetime of a project?

Section II developed the concepts of forward time dependence in order to identify foundational periods of development. Later periods all build on these major changes. Identification of foundational periods is crucial for understanding which phases of the software development process have to be understood very well.

To find out in Figure 2 to which quarter the backward time dependencies of age 2 in cell (3,2) point, it suffices to follow the diagonal line of cells crossing (3,2) in the lower-left direction, because backward dependencies of age 2 in quarter 3 (3,2), stem from the same quarter as the backward dependencies of age 1 in quarter 2 (2,1), i.e. the inner-period dependencies of quarter 1 (1,0). If we turn this reasoning the other way around, we can find all quarters that build on the changes made in a given quarter by following the diagonal line of cells crossing that quarter in the upper-right direction. In other words, the dark diagonal lines in the heatmap stem from the foundational periods.

The longer and the darker the diagonal, the more foundational a quarter is, i.e. later quarters keep on building on the changes made in that quarter. Figure 3a and Figure 3b show some explicit diagonal lines, but also white regions (similar to Figure 2). These regions correspond to quarters that are no longer foundational for later quarters. In PostgreSQL, for example, diagonal lines originating from quarters 10 to 13 suddenly stop at quarters 23 and 24 (Summer of 2002). In this quarter, important parts (e.g., plug-ins and tools) of the source code were extracted from the PostgreSQL code base and moved to the GBorg (now: PgFoundry) PostgreSQL community repository. Globally, FreeBSD has more light periods than PostgreSQL. We discuss our findings below.

The first foundational period in PostgreSQL and FreeBSD is the most foundational

The heatmaps in Figure 3a and Figure 3b of both PostgreSQL and FreeBSD have one or two very long diagonals early on in their life. To better analyze the diagonal lines in the heatmaps, we measured for each quarter its impact on later quarters, i.e. the total number of forward time dependencies in a quarter. This corresponds to the sum of the number of backward time dependencies along the cells in each diagonal of the heatmaps. This data is plotted in Figure 6a and Figure 6b. Foundational quarters show up as peaks in these graphs.

For PostgreSQL, the most foundational periods are quarters 1 and 2, for FreeBSD this is quarter 5. PostgreSQL started in 1996 from the Postgres95 1.01 source code, which had been made available as open source. The first half year of PostgreSQL's life, was spent on significantly cleaning up and restructuring the source code (e.g. the header file directories), culminating in the release of PostgreSQL 6.0 in January 1997. After this cleanup, the core of PostgreSQL did not change that drastically anymore, resulting in two foundational periods. The influence of FreeBSD's quarter 5 stems from the 4.3BSD-Lite lawsuit we discussed in Q2. Ten year later, there are still many changes depending on changes from these quarters. Every developer needs a good understanding of these quarters and the code changes in them to reliably and confidently make changes up until today.

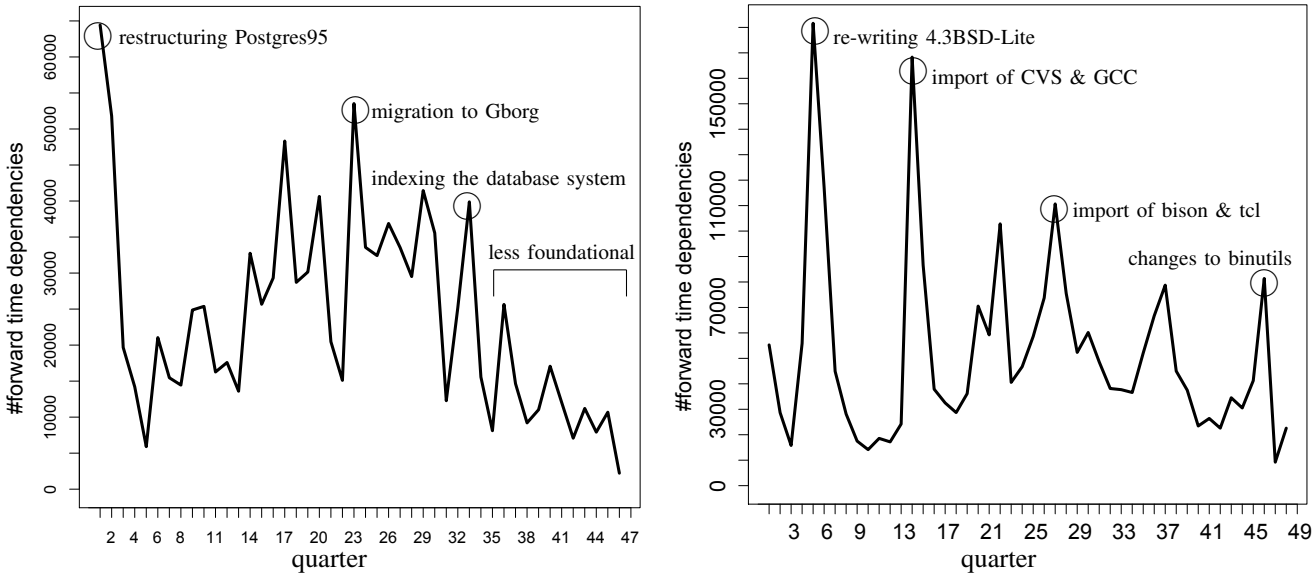
Both the PostgreSQL and FreeBSD heatmaps show a sequence of short diagonals following the diagonal of their first foundational quarter. Manual analysis shows that these short diagonals correspond to periods of polishing of less crucial parts of the source code.

Quarters are foundational because of large code imports or invasive changes

Figure 6a and Figure 6b show that PostgreSQL and FreeBSD both have peaks, but that FreeBSD has much more pronounced foundational quarters, i.e. the dark regions in the heatmap of Figure 3b contrast more with the white regions. This again confirms the periodic nature of time dependencies in FreeBSD.

For PostgreSQL, quarters 1 and 2 were highly foundational (as mentioned earlier). In quarter 23 a large piece of code was duplicated in preparation of the migration to the GBorg repository, whereas peaks near quarter 25 were responsible for the actual migration and important changes of key libraries and client program interfaces. Quarters 29 and 33 saw important changes to the database indexing system and the introduction of tablespaces. It is also noticeable from Figure 6a that PostgreSQL develops less foundational periods starting from quarter 34, when it starts to build on older periods as discussed in Q2 and shown in Figure 5a.

FreeBSD's first foundational quarter (quarter 5) has been discussed earlier. Similar to Q2, the most foundational quarters coincide with the imports of large, external chunks of source code into the base system. Quarters 14 and 22 saw the import of CVS, GCC, sh, bison, tcl and Perl. Quarter 43 contained "device mega-patches" (changes to drivers), important changes to the locking of per-process resource limits and the massive modification of many source code



(a) PostgreSQL

(b) FreeBSD

Figure 6. The total number of forward time dependencies in quarters for PosgreSQL and FreeBSD.

files. Finally, quarter 46 saw huge changes to the binutils, gdb, GCC, libstdc++ and sendmail versions in the base system.

Large code imports or invasive changes occur in foundational quarters. PostgreSQL's very first quarters are very foundational, whereas it took FreeBSD a few quarters to create its foundational structure.

V. Limitations and Future Work

Our case study is based on two open source projects, so our findings may not generalize to commercial projects, as open source projects have different characteristics. In addition, our findings might not generalize to open source projects in other domains, although we considered two systems from different domains to counter this threat to validity.

We calculate time dependencies for each quarter. Other periods (e.g., monthly, yearly, or per release) could be used and are likely to lead to other interesting results.

This paper considers "foundational periods" as those periods with the highest spikes in the number of forward time dependencies. However, managers and people who have better understanding of the projects could explicitly define "(non-)foundational periods" for their individual projects in terms of a threshold for the number of forward time dependencies.

Our time dependence relations are derived from static code dependencies. Implicit dependencies due to dynamic dependencies are not considered. Because of this, we would miss some of the time dependence relations.

We only look into one level of call graph dependencies, i.e. the direct dependencies of an entity. Dependencies below the first level are not considered in our study, because the impact of changes to these dependencies is smaller due to information hiding [11]. We would like to explore this assumption in future work.

In addition, we would like to study the relation between core architectural components of software projects and foundational periods. We would like to discover whether or not architectural cores appear in foundational periods. This could allow to identify architectural cores based on foundational periods.

VI. Related Work

In this section we discuss related work to this paper. In general, research in software evolution [12], [1], [2] and software metrics [13], [14] detects or monitors development periods and areas with slow or rapid growth. However, these approaches examine the final outcome (the changes) instead of exploring the temporal dependence of these changes.

Other researchers studied static dependencies between software entities like classes and methods to predict change coupling, i.e., what other part of the code needs to be changed if a given piece of code is changed [15], [16]. However, the dependencies studied 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 its call graph entities.

Several researchers used historical data to understand long-lived software systems. Chen et al. [17] introduced CVSSearch, a tool that tracks the fragments of source code by using CVS comments. Mockus et al. [18] classify maintenance activities by studying the textual description of a change. They validate their study by comparing the result of their automated approach with opinions of developers. Hassan and Holt [4] attach *Source Sticky Notes* to a static dependency graph to better understand the architecture of software systems. Our approach of tracking time dependence of source code changes leverages historical data to understand the foundational periods on which the development activity in a period is built.

We introduced the concept of time dependence in previous work [10] to assist managers in tracking the progress of a project. For this, we only had to consider the most recent backward time dependency of all changes in a period instead of considering *all* backward and forward time dependencies of an entity as done in this paper. By only considering the recent backward time dependency, one cannot find the foundational periods as those periods are defined in terms of all future periods that build on it. Hence the need to establish time dependence relations between a change and all prior changes it builds on. Our prior work also explored the impact of time dependencies on the appearance of bugs. Studying the relation between foundational periods and appearance of bugs is left as future work.

Brudaru and Zeller propose to measure the genealogy of changes [19]. They use a directed acyclic graph to model the impact of changes on defects at the level of lines of code. This information is used to analyze the future impact of changes on defects, development effort and maintainability of a system. Change dependencies are obtained by iteratively building the system without a change and then observing which changes are broken, although alternative heuristics are explored in their approach. Our approach makes use of readily available information to build our time dependence relations at the entity-level instead of the line-level.

The closest work to this paper is done by German et al. [20]. They propose the concept of a Change Impact Graph (CIG) to detect the impact of dependent changes when changing a source code entity. They visualize the call graph of a function and call graphs of its called entities iteratively within a time window. German et al. use their approach primarily to locate defects of a function. Our approach is not directly aimed at assisting developers during their daily development tasks. Rather, it assists managers and researchers in understanding which periods have provided the structural foundation on which later periods build.

VII. Conclusion

Building software has many similarities with physical construction, with new changes often building on older changes. Throughout the lifetime of a project, there exists foundational periods during which critical code changes are done. Such code changes create the foundational structure on which other code changes and periods build. A good understanding of this hierarchy of temporal dependence is needed for managers to better plan their projects and for researchers to study the evolution of long-lived projects. Knowing such foundational periods, managers should ensure that either the documentation about these periods is up-to-date or that sufficient human expertise is available about these periods.

Through a case study on two large open source systems, we made the following important findings: 1) On average, up to 48% of the changes done in a period build on changes in the same development period (i.e., quarter). 2) As projects age and become more stable, they either progressively build on older periods or cycle between building on old and new periods. 3) Invasive refactorings and major code imports are done during foundational periods, with earlier foundational periods having a stronger impact on the other periods.

Acknowledgments

We would like to thank the anonymous reviewers for their valuable comments.

References

[1] M. W. Godfrey and Q. Tu, "Evolution in open source software: A case study," in *ICSM '00: Proceedings of the International Conference on Software Maintenance (ICSM'00)*. Washington, DC, USA: IEEE Computer Society, 2000, p. 131.

[2] 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 *METRICS '97: Proceedings of the 4th International Symposium on Software Metrics*. Washington, DC, USA: IEEE Computer Society, 1997, p. 20.

[3] A. E. Hassan, "Mining software repositories to assist developers and support managers," Ph.D. dissertation, University of Waterloo, Waterloo, ON, Canada, 2004.

[4] A. E. Hassan and R. C. Holt, "Using development history sticky notes to understand software architecture," in *IWPC '04: Proceedings of the 12th IEEE International Workshop on Program Comprehension*. Washington, DC, USA: IEEE Computer Society, 2004, p. 183.

[5] A. E. Hassan, "Automated classification of change messages in open source projects," in *SAC '08: Proceedings of the 2008 ACM symposium on Applied computing*. New York, NY, USA: ACM, 2008, pp. 837–841.

[6] "PostgreSQL official website," http://www.postgresql.org/.

[7] "FreeBSD official website," http://www.freebsd.org/.

[8] A. E. Hassan and R. C. Holt, "The chaos of software development," in *IWPSE '03: Proceedings of the 6th International Workshop on Principles of Software Evolution*. Washington, DC, USA: IEEE Computer Society, 2003, p. 84.

[9] K. Bennett, "Legacy systems: Coping with success," *IEEE Software*, vol. 12, no. 1, pp. 19–23, 1995.

[10] O. Alam, B. Adams, and A. E. Hassan, "Measuring the progress of projects using time dependence of code changes," in *ICSM '09: Proceedings of the 25th IEEE International Conference on Software Maintenance (ICSM'09)*, 2009, to Appear.

[11] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Commun. ACM*, vol. 15, no. 12, pp. 1053–1058, 1972.

[12] H. Gall, M. Jazayeri, and J. Krajewski, "CVS release history data for detecting logical couplings," in *IWPSE '03: Proceedings of the 6th International Workshop on Principles of Software Evolution*. Washington, DC, USA: IEEE Computer Society, 2003, p. 13.

[13] S. Bouktif, G. Antoniol, and E. Merlo, "A feedback based quality assessment to support open source software evolution: the grass case study," in *ICSM '06: Proceedings of the 22nd IEEE International Conference on Software Maintenance*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 155–165.

[14] P. M. Johnson, H. Kou, M. Paulding, Q. Zhang, A. Kagawa, and T. Yamashita, "Improving software development management through software project telemetry," *IEEE Softw.*, vol. 22, no. 4, pp. 76–85, 2005.

[15] S. Mirarab, A. Hassouna, and L. Tahvildari, "Using bayesian belief networks to predict change propagation in software systems," in *ICPC '07: Proceedings of the 15th IEEE International Conference on Program Comprehension*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 177–188.

[16] Y. Zhou, M. Würsch, E. Giger, H. C. Gall, and J. Lü, "A bayesian network based approach for change coupling prediction," in *WCRE '08: Proceedings of the 2008 15th Working Conference on Reverse Engineering*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 27–36.

[17] A. Y. Yao, "Cvssearch: Searching through source code using cvs comments," in *ICSM '01: Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*. Washington, DC, USA: IEEE Computer Society, 2001, p. 364.

[18] A. Mockus and L. G. Votta, "Identifying reasons for software changes using historic databases," in *ICSM '00: Proceedings of the International Conference on Software Maintenance (ICSM'00)*. Washington, DC, USA: IEEE Computer Society, 2000, p. 120.

[19] I. I. Brudaru and A. Zeller, "What is the long-term impact of changes?" in *RSSE '08: Proceedings of the 2008 international workshop on Recommendation systems for software engineering*. New York, NY, USA: ACM, 2008, pp. 30–32.

[20] D. M. German, G. Robles, and A. Hassan, "Change Impact Graphs: Determining the Impact of Prior Code Changes," in *International Working Conference in Source Code Analysis and Manipulation (SCAM) 2008*, Sept. 2008.