Measuring the Progress of Projects Using 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

Tracking the progress of a project is often done through imprecise manually gathered information, like progress reports, or through automatic metrics such as Lines Of Code (LOC). Such metrics are too coarse-grained and too imprecise to capture all facets of a project. In this paper, we mine the code changes in the source code repository and study the concept of time dependence of code changes. Using this concept, we can track the progress of a software project as the progress of a building. We can examine how changes build on each other over time and determine the impact of these changes on the quality of a project. In particular, we study whether new changes are built just-in-time or if they build on older, stable code. Through a case study on two large open source projects (PostgreSQL and FreeBSD), we show that time dependence varies across projects and throughout the lifetime of each project. We also show that there is a high linear correlation between building on new code and the occurrence of bugs.

1. Introduction

Getting accurate and timely feedback about the progress of a software project is critical to deliver high quality results on time and within budget [8, 11]. For this, project managers often use informal meetings with the development team and manually compiled progress reports. However, these meetings and reports do not provide sufficient feedback, causing many projects to fail [31, 32]. Most of these failures are attributed to the discrepancy between the development process and project management [4, 21]. The act of obtaining accurate and timely progress about which development activities are on time, which ones are delayed and which activities could be rescheduled to resolve a conflict, has not been studied widely before.

We propose to track a project as one would track the construction of a building, with each change providing the structure on which other changes can build. As with buildings, some changes can build on fresh structure (*built-on-new changes*) while other changes can build on well-established structure (*built-on-old changes*). Furthermore, some changes might build on no structure at all (*independent changes*). As a construction evolves, one must ensure that structure is being put just-in-time to avoid costly development that might never be used or that might not be needed for a while (and hence only add complexity). Furthermore, engineers are often concerned about the risks of building on fresh structure. Such analogies have never been explored before in relation to the construction of software systems.

We believe that the required information for evaluating the progress of a project is readily available in the source code repository of a project. The repository contains the building blocks of software features in the form of changes to functions and files. The repository also contains information about the temporal dependence between these changes. We propose the concept of a time dependence relation between source code changes. A time dependence relation between two changes indicates that one change to a source code entity follows (depends on) another change. Using these relations, we can track not only the location of a change but also the changes it depends on.

To show that time dependence is a promising technique for ensuring accurate and timely tracking of the progress of a project, we apply the technique to two large open source systems with a long development history, i.e., PostgreSQL and FreeBSD. Our case study shows that:

- Over 50% of all changes in a quarter are built-on-new changes. The rate of built-on-new changes, built-on-old changes and independent changes varies throughout the lifetime of a project and across projects. Large fluctuation in the rate is indicative of major structural changes and feature additions in the studied projects.

- The regular development and bug fix processes follow the same breakdown of change types (built-on-new, built-on-old and independent). This is a surprising finding, since we would have expected that the bug fix process would be more dependent on recent activ-

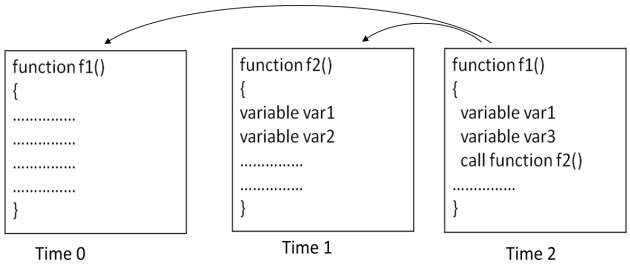

Figure 1. Time dependence between changes.

ities in a project, relative to the regular development process.

- Spending more of the *regular* (i.e., non-bug fix) development time budget building on new changes requires spending more of the *total* development time on fixing bugs. This finding matches well with common construction wisdom concerning building on fresh structure compared to building on long-established structure.

The paper is organized as follows. Section 2 presents the concept of time dependence and discusses how it allows to measure the project progress. Section 3 presents the four research questions we want to address using time dependence. In Section 4, we explain the setup of the two case studies we performed, and we discuss case study results for each research question. Limitations and future work are described in Section 5, related work is presented in Section 6, and our conclusions are summarized in Section 7.

2. Time Dependence

The problem of measuring the progress of a software project boils down to measuring the progress of source code changes. In an ideal project, changes should occur just as they are needed (i.e., just-in-time). Changes (i.e., structures) that are not needed for a few months do not need to be put in a particular quarter. Instead, these changes can be pushed forward and other changes could be brought in. In a continuously evolving project, we expect that a large number of changes will build on recently added code instead of depending on old, unchanged code, unless a major restructuring happens on well-established structures within the software system. While traditional views of software evolution track basic metrics like lines of code (LOC) [15, 26], we propose the concept of time dependence to quantify our intuition about the progress of a project.

Time dependence captures the time between "related" source code changes and records which changes are independent of other changes. In particular, a time dependence relation measures the time gap between a change of a source code entity E (like a function or type definition) at time T, and the last change of E and of each entity which E depends on. The example in Figure 1 illustrates this concept. Function `f1` is added at *time 0*. Function `f2` is added at *time 1*. At *time 2*, `f1` is modified to add a call to `f2`. We say that `f1` at *time 2* depends on the change (introduction) that happened to `f2` at *time 1* and on the most recent change to `f1` *time 0* (in this case the introduction of `f1`). These prior changes provided the structure needed for the change at *time 2*. This structure could be conceptual (i.e., a higher-level feature) or concrete (i.e., just implementation/code).

Each change can depend on a large number of prior changes, but for this paper we consider the smallest time interval between a change and any of its dependent changes. Using the example in Figure 1, we would say that the change to `f1` at *time 2* depends on the change at *time 1*. This indicates that the earliest possible time to perform the change of *time 2* is *time 1*, since at that time all the needed structure was in place. We note that this formulation does not account for the complexity of software development in relation to features. For example, consider the case of `f1` being the cut-and-paste logic in an application, with the change at *time 2* introducing the ability to cut-and-paste HTML code. Although *theoretically* the change could have been done at *time 1*, it might be the case that at *time 1* HTML was not a popular format worth supporting yet. In short, our formulation does not factor in the requirements of a project and the external factors that drive changes. However, our formulation provides information to practitioners about *possible delays and dependencies*. Practitioners need to examine these findings in the context of their project and their expectations for a particular change.

To provide a high-level view of the progress of a project, we study all changes in a particular period (e.g., month, quarter, year). We could even study the time dependence of changes across releases, though we do not perform this study in this paper. Whatever approach we use, we must "lift" all time dependence relations within a particular period. Figure 2 illustrates the lifting process. Three periods are shown on Figure 2a, each of these grouping the changes that happened inside them. All edges correspond to the time dependence relation from a change to its most recently changed dependent change. This relation either links changes inside the same period (dashed edge) or between different periods (full edge). To lift the time dependence relations up to the period level, we introduce the concepts of "built-on-new", "built-on-old" and "independent" changes:

built-on-new change depends on a change in a recent period.

built-on-old change depends on a change in an older pe-

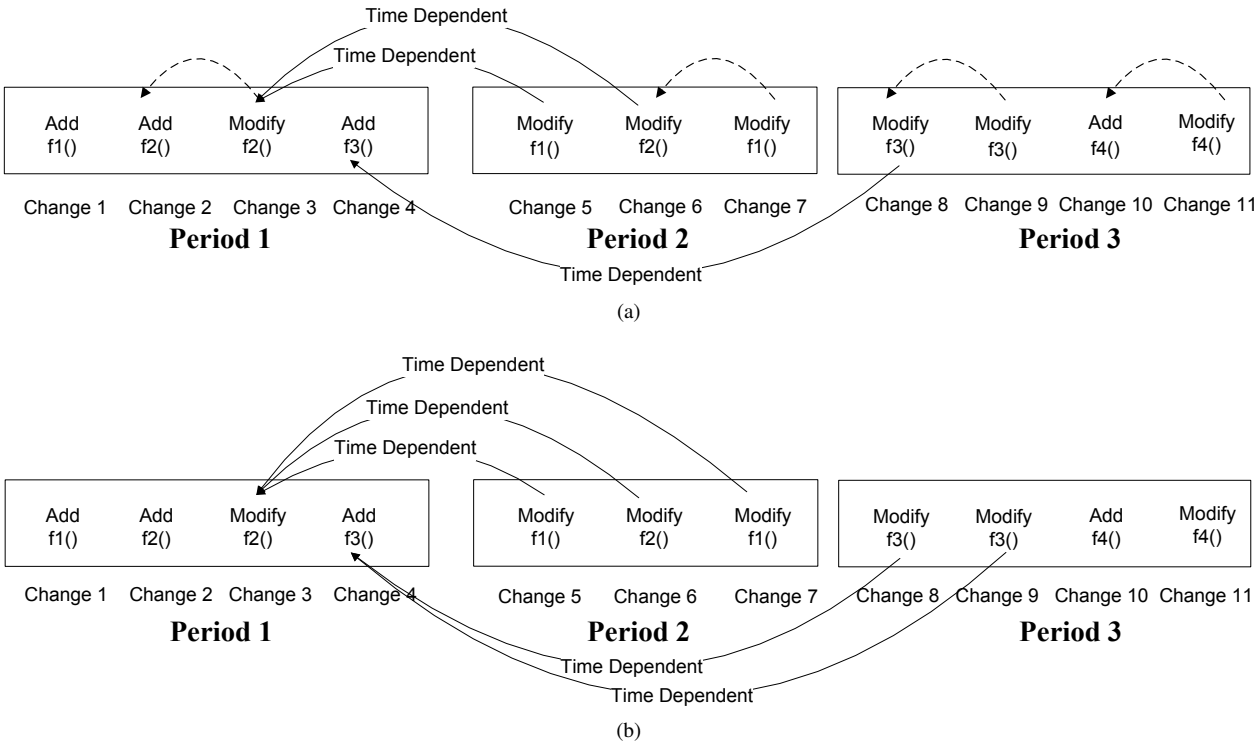

Figure 2. Time dependence between periods (a) before and (b) after resolving transitive dependence.

riod.

independent change does not depend on changes in any prior period.

Figure 2b shows the actual time dependence relations of each change after resolving the transitive changes within each period, i.e., after replacing each dashed edge with a dependency on a change of an immediately preceding period (built-on-new), a dependency on an older change (built-on-old) or no dependency at all (independent). Changes 10 and 11 are clearly independent. If we consider "built-on-new" to mean "builds on changes in the last period", then changes 5, 6 and 7 are built-on-new, whereas changes 8 and 9 are built-on-old. In one of our experiments, we determine what the right number of periods is to consider a change as built-on-new.

3 Research Questions

Using the concept of time dependence, we sought to study four research questions. The first two questions are about the nature of tracking the progress of projects, whereas the last two questions address two common project management beliefs. We briefly state and motivate the relevance of each question:

Q1 How does the time dependence of changes vary over time?

Do projects primarily build on recently modified source code (built-on-new), or is there a gap between a change and its most recent dependent change? As a project ages, does its development progress slow down with less built-on-new changes and more built-on-old changes? Using our dependence analysis and their knowledge of a project, practitioners can monitor more closely the progress of a project.

Q2 What is the impact of independent changes?

Independent changes do not have timing constraints. Such changes, in theory, can be re-scheduled and moved around without any problems. However, it is not clear whether such changes are common in systems. Are some systems more open to independent changes than others?

Q3 Is the distribution of time dependence similar for regular development and bug fix processes?

Maintenance activities like bug fixing have been identified before as having a negative impact on developer productivity, and hence on the progress of a software project [32]. However, do bug fix changes and

	PostgreSQL	FreeBSD
type	DBMS	Operating System
period	1997–2007	1994–2005
#changes	84,311	353,958
#entities	31,863	253,896
#files	2,053	21,093
#Bug fixes	22,913	32,074

Table 1. Studied systems' characteristics.

enhancement changes exhibit the same distribution of time dependence relations, or are bug fixing and regular development completely independent of each other? This information could help project managers derive information about the progress of the bug fix process from the progress of the regular development process, and vice versa. For example, do regular development and bug fixing require similar operational knowledge about a system?

Q4 Is building on recent changes risky?

Construction workers do not build a house on a fresh foundation, instead they wait for the foundation to dry up to not risk the collapse of the house. Does developing on built-on-new changes lead to more bugs in software? Having this information could be used to decide how many resources should be allocated to testing.

4. Case Study

To explore our four research questions, we performed a case study using two large long-lived open-source projects. We briefly present the used data and systems, then we present the results for each one of our four questions.

4.1 Studied Systems

We used data from the open source PostgreSQL (1997–2007) and FreeBSD (1994–2005) projects for our case study. PostgreSQL is a relational database system of which the original design goes back to the 1980s [30], whereas FreeBSD is an operating system distribution derived from the Berkeley flavor of UNIX [13]. We studied the FreeBSD base system, without the ports and applications. Table 1 shows the number of changes and files for both projects. We picked both systems due to their long history of changes, which is easily accessible. The two systems being from two different domains (databases and operating systems) would help us verify the generality of our findings across domains.

For each system, we extract the history of changes from the source code repository at the level of entities. We

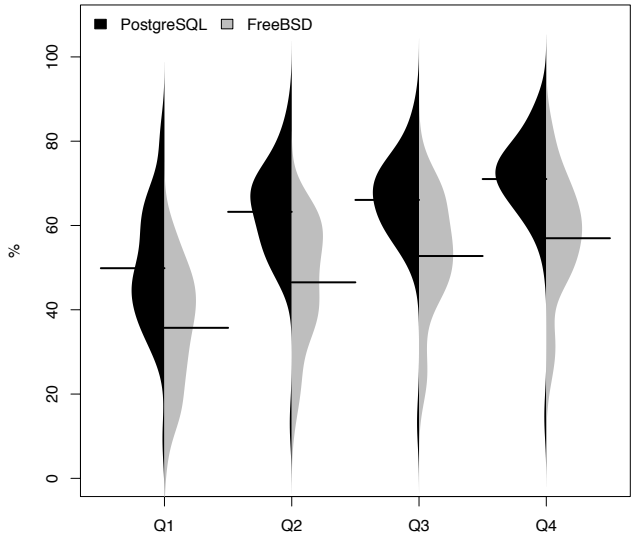

Figure 3. Beanplots with the percentage of built-on-new changes for different quarter delays.

lift line-level change information to the level of code entity changes, like functions, function calls and variable accesses [17]. Using this entity-level information, we can build the time dependence relation for each source code change. The resulting information can be aggregated to describe the progress of a project at the level of periods (cf. Figure 2a), or used as-is to identify changes with particularly high delay or changes that follow each other very closely. Being able to identify specific changes, we could map our high-level observations about the progress of a project to concrete examples of changes. Using these concrete examples, practitioners can verify high-level observations using their experience and read through the metadata attached to each change [18]. Possible metadata includes the change commit message and the name of the developer responsible for the change. With this data, we can distinguish between ***bug fix changes*** and other, non-bug fixing ***enhancement changes*** [19]. We also filter out changes that are done to indent the code or to update copyright notices, since these are not interesting for our analysis.

We present below the results of our four research questions.

Q1. How does the Time Dependence of Changes Vary over Time?

To define the concept of built-on-new and built-on-old changes for our studied systems, we measure the percentage of changes that are built on the last one, two, three and four quarters. These percentages are represented in

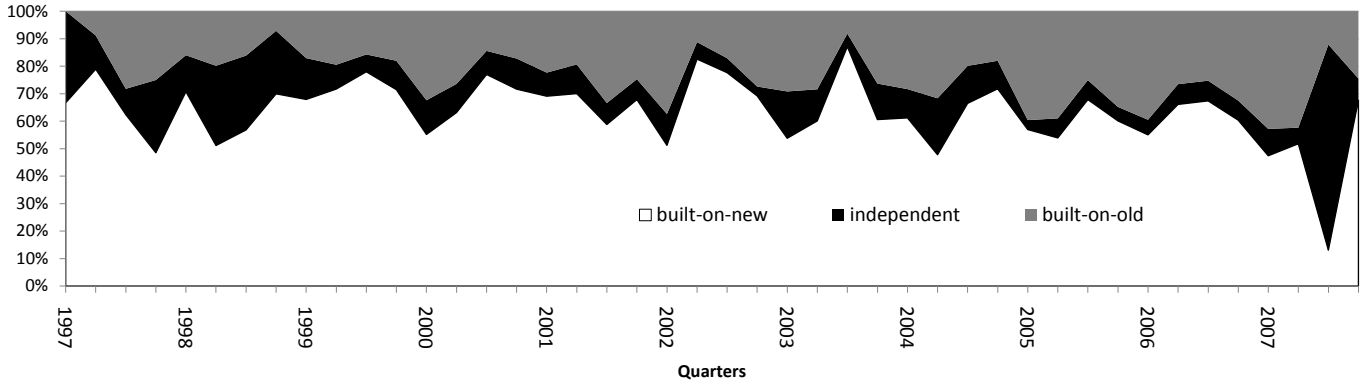

Figure 4. The distribution of built-on-new, built-on-old and independent changes (PostgreSQL).

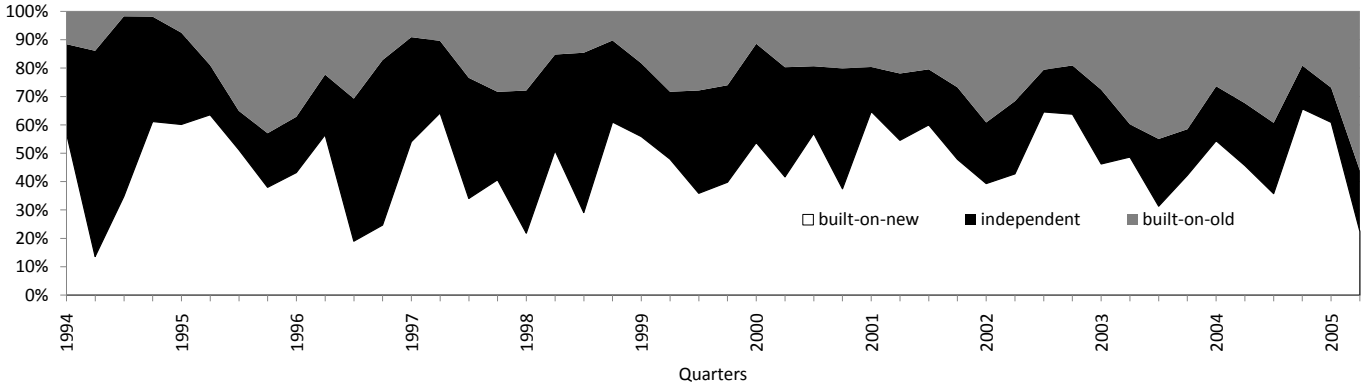

Figure 5. The distribution of built-on-new, built-on-old and independent changes (FreeBSD).

Figure 3 as four beanplots [24], each beanplot comparing the data for PostgreSQL (left) and FreeBSD (right). A beanplot contains more information than a boxplot, as its width shows how the observations for each quarter are distributed over the possible values (in this case the percentage of changes). This facilitates the comparison of the PostgreSQL and FreeBSD distributions. The horizontal black lines show the average of each distribution.

From the beanplots, we note that the growth of the percentage of built-on-new changes slows down above two quarters. For PostgreSQL, on average 49.8% of the changes in a quarter depend on the last quarter, whereas on average 63.2% depend on the last two quarters. For FreeBSD, the averages are 35.7% and 46.5%, respectively. When moving to periods longer than two quarters (e.g., from two to three quarters), we only gain a small increase, less than 10%. Similarly, the 25th and 75th percentiles of each distribution increase less than 10% when moving to periods longer than two quarters. For this reason, we chose two quarters as the boundary between built-on-new and built-on-old changes. The beanplots show that PostgreSQL depends much more on recent changes than FreeBSD. In particular, the average percentage of built-on-new changes for PostgreSQL is more than 15% higher than for FreeBSD and the beanplot density is shifted up considerably compared to FreeBSD.

Figure 4 and Figure 5 show cumulative plots of the percentages of built-on-new, built-on-old and independent changes for PostgreSQL and FreeBSD respectively. A larger area means that the corresponding type of change occurs more frequently. Overall, there are many fluctuations in the distribution of the three kinds of changes, and no clear upward or downward evolution can be observed. Built-on-old changes (grey area) seem to be equally important for PostgreSQL and FreeBSD. Built-on-new changes (white area) take up the majority of changes for PostgreSQL, whereas for FreeBSD the development seems to be spread more evenly across all three types of changes. In particular, there is a large number of independent changes in FreeBSD over time, possibly due to architectural characteristics of FreeBSD over PostgreSQL. The large number for FreeBSD might be due to its independent code bases for hardware drivers and for externally developed system tools like GCC or standard libraries that were imported into the FreeBSD base system, whereas the “contrib” code

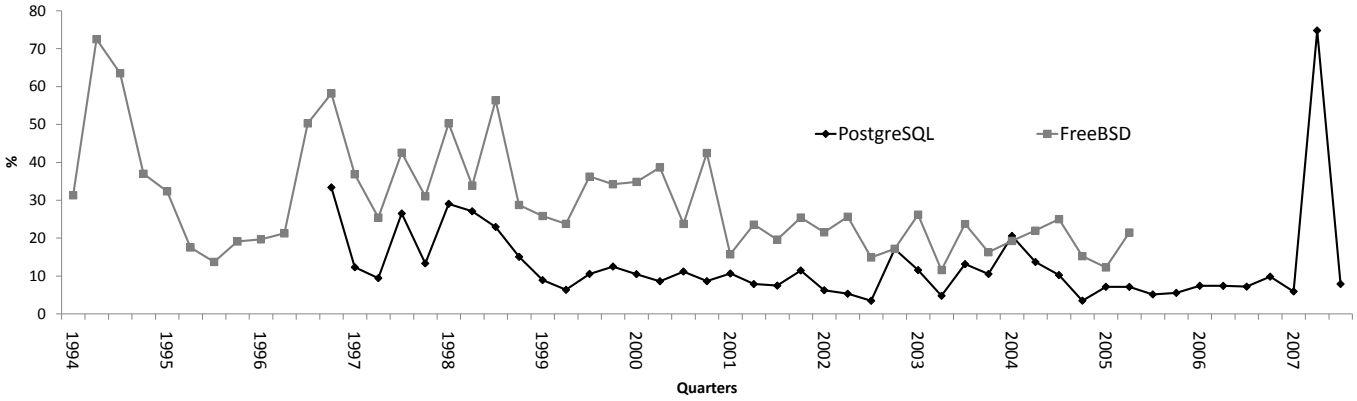

Figure 6. Percentage of independent changes in PostgreSQL and FreeBSD.

base for PostgreSQL-related tools has a much smaller scale. There are only a few periods for PostgreSQL (2007) and for FreeBSD (1994 and 1996) where the percentage of built-on-new changes drops below 20%. Those periods saw an unusually high number of independent changes (more on this in the following question).

We manually examined all changes in 2003 for PostgreSQL and all changes in 1997 for FreeBSD, as both years saw a high percentage of built-on-new changes. For each type of change (built-on-new, built-on-old and independent), we determined those entities that were changed most frequently. We then examined these entities and read through the change messages attached to these changes. We report here on prominent examples of built-on-new and built-on-old changes we found. The next section (Q2) gives examples of important independent changes.

For the studied period in PostgreSQL, we found that all 138 changes to the "ecpg" compiler for Embedded SQL were done just-in-time, with each change depending on recently added changes. The changes comprised the move to a new GNU Bison parser generator, changes to the build system, the addition of an Informix compatibility mode and various bug fixes. We also found examples of built-on-old changes, such as the addition of support for a new version of the message protocol between the PostgreSQL front and back end (out of 43 changes, 27 were built-on-old changes).

For the studied period in FreeBSD, we note one example of built-on-new changes, and two examples of built-on-old changes. A significant set of built-on-new changes merged SMP support (Symmetric MultiProcessing) for the amd64 and i386 architectures into FreeBSD. This was clearly a major effort, as 90 existing files were modified, many of them at the core of the kernel, and 11 new files were added. These changes were really just-in-time. A first example of built-on-old changes, was the addition of timeout support to the sysinstall installation utility, with some changes building on changes that were done almost three years earlier. The

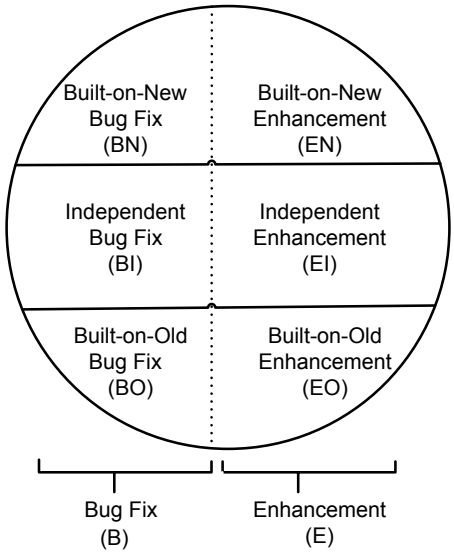

Figure 7. Categories of changes.

second example involved changes to the internationalization code to add support for the Japanese language. These changes built on changes that were done more than one year earlier.

Time dependence varies across projects and throughout time. Changes in the FreeBSD project are evenly distributed across the three types of changes (built-on-new, built-on-old and independent), whereas changes in PostgreSQL more frequently build on newer changes.

Q2. What is the Impact of Independent Changes?

Independent changes are "floating" changes which do not depend on changes from prior quarters. These changes could have been made much earlier, if the requirements for these changes were known in advance and resources were available. Hence, it is interesting to compare the distribution of independent changes for PostgreSQL and FreeBSD. To make this comparison a bit easier, Figure 6 plots the black areas of Figure 4 and Figure 5 on one graph.

Figure 6 shows that FreeBSD has a much higher percentage of independent changes than PostgreSQL. Since we studied the FreeBSD base system, it contains a kernel, device drivers and system tools like compilers and libraries. Device drivers are self-contained modules with dedicated logic for supporting devices like hard disks and network cards. As a clean plug-in mechanism exists for drivers, new drivers show up as independent changes. Similarly, system tools introduce independent changes. Peaks in the percentage of independent changes coincide with the import of large chunks of source code of externally developed tools like CVS, GCC, sh, Bison and Perl for customization. The high percentage of independent changes shows that the architecture of FreeBSD overall has better support for extensibility and independent development over PostgreSQL.

We note that in the second half of 2007, PostgreSQL experienced a huge spike in the percentage of independent changes. We verified this finding by contacting the PostgreSQL developers. These explained that two large independent features had been added to the back end in August and September 2007, in preparation of the 8.3 release [10]:

- Tom Lane, a major PostgreSQL developer, migrated the *Tsearch2* functionality ("Text Searching") from the separate "contrib" (see Q1) directory into the core directories. The Tsearch2 feature enables the searching of terms in natural-language documents.

- One month later, the same developer committed large additions to the *HOT* subsystem ("Heap Organized Tuples"), which is an optimization feature for throughput and more consistent response time.

The large percentage of independent changes shows that FreeBSD imports large chunks of external source code and that the architecture of FreeBSD is built for extension, whereas spikes of independent changes in PostgreSQL signal sudden additions of large independent features.

	PostgreSQL	PostgreSQL*	FreeBSD
built-on-new	0.64	0.87	0.81
built-on-old	0.85	0.9	0.91
independent	0.57	0.84	0.9

Table 2. Pearson correlations between the relative percentage of built-on-new, built-on-old or independent bug fix and enhancement changes in a year for PostgreSQL and FreeBSD. The * correlations do not take into account the last year (2007).

Q3. Is the Distribution of Time Dependence Similar for Regular Development and Bug Fix Processes?

Software development activities consist of two processes: the regular development process and the bug fix process. Developers devote their time across both processes, with bug fixing interleaved with regular development. We would like to study the relation between these two related, yet different, processes. Do both types of processes build on similar change periods or do they differ? For example, are there periods with bug fixing depending on old changes while regular development depends on newer changes? If both processes depend on similar change periods, then one can reduce the negative impact on the productivity of developers of interleaving bug fixing with regular development [32].

Using our time dependence formulation, this problem boils down to analyzing possible correlation between the three types of enhancement (built-on-new, built-on-old and independent) with the same types of bug fixes, i.e., we measure the correlation between $\frac{EN}{E}$ and $\frac{BN}{B}$, $\frac{EI}{E}$ and $\frac{BI}{B}$, and $\frac{EO}{E}$ and $\frac{BO}{B}$ in Figure 7. Do periods with a large percentage of built-on-new enhancements have a large percentage of built-on-new bug fixes, leading to a positive correlation between both processes, or do such periods have a small percentage of built-on-new bug fixes, leading to a negative correlation?

We measured the Pearson correlation values between the relative percentage of built-on-new, built-on-old and independent bug fix and enhancement changes for PostgreSQL and FreeBSD, at the year level (see Table 2). We calculated the correlations at the year level, since the number of bug fix changes per quarter is too small to make reasonable statistical claims. Using the same approach as discussed for Q1, we use one year as the boundary between built-on-new and built-on-old changes.

If we consider all the years for PostgreSQL and FreeBSD (columns one and three in Table 2), PostgreSQL obtains weak to strong correlations for the distribution of time de-

pendence for enhancement and bug fix changes, whereas FreeBSD obtains high correlations throughout.

Overall, PostgreSQL has a high linear correlation, except for the last year (2007). If we ignore 2007, and recompute the Pearson correlations (second column of Table 2), PostgreSQL shows much higher correlation numbers, even higher than for FreeBSD. We suspect that this is caused by the unusual spike in independent changes in 2007 (see Q2). The high correlation between built-on-new, built-on-old and independent bug fix and enhancement changes shows that developers need knowledge about similar periods of the development history when doing regular development and bug fixing.

Regular development and bug fixing require knowledge about source code changes from the same development periods, i.e., the context switch between both processes is relatively limited.

Q4. Is Building on Recent Changes Risky?

Construction workers avoid building on newly laid structure. Instead, they prefer to build on established dry structure. Workers fear that the new structure will be much riskier and will lead to problems. We wish to verify this common wisdom in the construction industry with software development. Using our time dependence formulation, we measure if between two years an increase in the number of *built-on-new* enhancement changes relative to *all enhancement* changes would lead to an increase in the number of *bug fix* changes relative to *all* changes, i.e., we measure the correlation between $\frac{EN}{E}$ and $\frac{B}{E+B}$ in Figure 7. In other words, if you spend more of your *regular development* time budget E to build on fresh changes, you are likely to spend proportionally more of your *total* time budget $E + B$ to fix bugs, and hence less of your *total* time budget $E + B$ to do regular development.

The Pearson correlation between both metrics turns out to be high: 0.65 for PostgreSQL and 0.91 for FreeBSD. The lower correlation for PostgreSQL can be explained by looking at Figure 8. This graph shows for each year the increase of the considered percentages compared to the previous year (negative increase means decrease). From 2000 to 2001, and from 2003 and 2004 an increase of the relative percentage of built-on-new enhancement changes was accompanied by a decrease of the total percentage of bug fix changes. Investigating what happened in these periods is future work.

Overall, the high linear correlation for PostgreSQL and FreeBSD suggests that managers should be careful when they plan to build relatively more enhancements on fresh structure (i.e., new changes). They might consider assigning extra resources on testing. Our experiment provides an empirical proof of what seems to be common wisdom in project management.

Building on recent changes is risky, as more of the development time is spent fixing bugs.

5. Limitations and Future Work

Our technique to recover the time dependence relations only takes into account static dependencies. Implicit dependencies due to dynamic dependencies are not considered. This leads to an overestimation of independent changes.

Our analysis requires interpretation by project experts to really determine whether a particular change could have been done earlier or whether the requirements did not exist at that earlier point in time. For our case study, we validated our approach by reading the documentation, manuals, repository logs and mailing lists, and by contacting developers in the studied projects.

Our case study was done on two open source projects, so our findings may not generalize to commercial projects, as open source projects have different characteristics related to facets like testing and communication. In future work, we would like to perform a user study to determine the benefit of this approach in a real project setting instead of just mining the repository of a project.

6. Related Work

We discuss existing work directly related to project scheduling and prediction of bugs, although there are also similarities with techniques for detecting software restructuring (e.g., [12]).

Research in software evolution [14, 15, 26] and software metrics [7, 22] 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 characteristics and temporal dependencies between these changes.

A large part of research in project scheduling tries to estimate the optimal project plan up-front (e.g., [1, 2, 23, 28]), such that the project's goals and constraints are satisfied with a minimal amount of risk. We on the other hand are interested in extracting sufficient feedback from the source code repositories to track the progress of a project, i.e., how well the original plan is followed. Still, source code repositories have been identified as an important data source for effort estimation as well [1, 28].

Kothari et al. [25] introduce the Change Cluster technique to track the evolution of software projects. They categorize the progress of a project into different areas or efforts, like maintenance and new development. The area

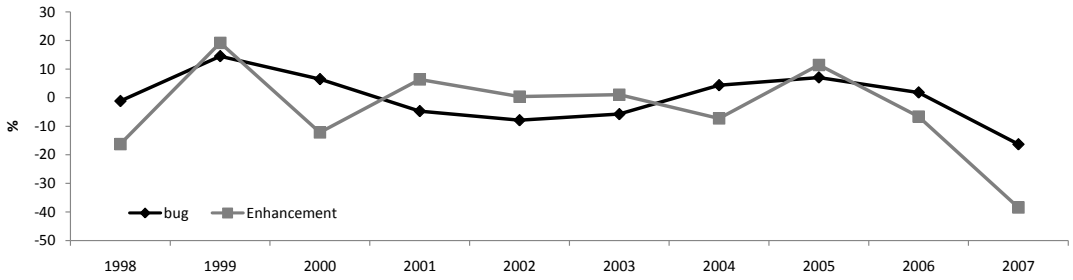

Figure 8. Graph showing the correlation between the yearly increase of the relative percentage of built-on-new enhancement changes and the yearly increase of the total percentage of bug fix changes (PostgreSQL).

of "new development" differs from our definition of independent changes, as it encompasses all newly added development, whether or not it depends on earlier features or changes. Therefore, their approach cannot identify the evolution of unconstrained changes.

Xing et al. [33] and Barry et al. [5] characterize the different periods of a software project using characteristics like spread of changes, effect of changes on dependency structure and number of changes. Their approaches cannot provide feedback either about how just-in-time enhancement or bug fixing changes have been made.

The closest related work to our paper is a proposal by Brudaru et al. to measure the genealogy of changes [9]. They model the impact between source code changes as a directed acyclic graph. Changes are studied at the level of lines of code, in order to to analyze the future impact of changes on defects, maintainability of a system and development effort. Change dependencies are obtained by iteratively building the system without a change and then observing which changes are broken, although alternative heuristics are explored for this approach. Our technique harvests time dependence relations directly from the information stored in the source code repositories to keep them up-to-date. By lifting this information up to the level of source code entities, time dependence relations are brought closer to the mind set of project managers.

The second group of related work consists of bug prediction techniques. Although some approaches detect bugs based on metrics like LOC [16, 20, 27] or on the presence of prior faults [34], the majority of recent techniques are based on information from the change history of a system, as extracted from the source code repository [3, 6, 16, 27, 29]. These techniques typically look at code churn [29] or the number of changes. For example, Bernstein et al. [6] use the number of revisions and reported issues in the last quarter to predict the location and number of bugs in the next month. In future work, we plan to compare the performance of our approach against approaches which use other types of historical data for predicting bugs.

7. Conclusion

We propose to study the process of building software as one would study the process of constructing a building, with new changes building on earlier changes. For this, we considered the temporal dependence between code changes. Such temporal dependence gives a different and interesting view of the progress of a software project, with changes building on fresh structure (i.e., built-on-new), changes building on old established structure (i.e., built-on-old), and changes not building on any previous structure (i.e., independent). Using these concepts, we studied two open source projects: PostgreSQL and FreeBSD. We found that 1) the time dependence varies across both projects and throughout time, 2) the architecture of a software application and its evolution impacts the observed time dependence relations, 3) both regular development and bug fixing processes tend to follow the same time dependence characteristics, and 4) spending more of the *regular development* time building on recent changes increases the *total* percentage of development time that has to be spent fixing bugs.

Through our approach, practitioners can track more closely the progress of their projects instead of having to depend on informal meetings and coarse-grained metrics.

Acknowledgments

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

References

[1] T. K. Abdel-Hamid and S. E. Madnick. The dynamics of software project scheduling. *Commun. ACM*, 26(5):340–346, 1983.

[2] E. Alba and J. Francisco Chicano. Software project management with gas. *Inf. Sci.*, 177(11):2380–2401, 2007.

[3] E. Arisholm and L. C. Briand. Predicting fault-prone components in a java legacy system. In *ISESE '06: Proc. of the 2006 ACM/IEEE international symposium on Empirical software engineering*, pages 8–17, New York, NY, USA, 2006. ACM.

[4] R. L. Baber. *Software Reflected: The Socially Responsible Programming of Computers*. Elsevier Science Inc., New York, NY, USA, 1982.

[5] E. J. Barry, C. F. Kemerer, and S. A. Slaughter. On the uniformity of software evolution patterns. In *ICSE '03: Proc. of the 25th Int. Conference on Software Engineering*, pages 106–113, Washington, DC, USA, 2003. IEEE Computer Society.

[6] A. Bernstein, J. Ekanayake, and M. Pinzger. Improving defect prediction using temporal features and non linear models. In *IWPSE '07: Ninth international workshop on Principles of software evolution*, pages 11–18, New York, NY, USA, 2007. ACM.

[7] 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: Proc. of the 22nd IEEE Int. Conference on Software Maintenance*, pages 155–165, Washington, DC, USA, 2006. IEEE Computer Society.

[8] F. P. Brooks, Jr. *The mythical man-month (anniversary ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

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

[10] J. Drake. Personal communication, January 2009. a developer of PostgreSQL.

[11] W. R. Duncan, editor. *A Guide To The Project Management Body Of Knowledge (PMBOK Guides)*. Project Management Institute, 2004.

[12] J. Ekanayake, J. Tappolet, H. C. Gall, and A. Bernstein. Tracking Concept Drift of Software Projects Using Defect Prediction Quality. In *MSR '09: Proc. of the 6th IEEE Working Conference on Mining Software Repositories*. IEEE Computer Society, May 2009. to appear.

[13] FreeBSD official website. http://www.freebsd.org/.

[14] H. Gall, M. Jazayeri, and J. Krajewski. Cvs release history data for detecting logical couplings. In *IWPSE '03: Proc. of the 6th Int. Workshop on Principles of Software Evolution*, page 13, Washington, DC, USA, 2003. IEEE Computer Society.

[15] M. W. Godfrey and Q. Tu. Evolution in open source software: A case study. In *ICSM '00: Proc. of the Int. Conference on Software Maintenance (ICSM'00)*, page 131, Washington, DC, USA, 2000. IEEE Computer Society.

[16] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy. Predicting fault incidence using software change history. *IEEE Trans. Softw. Eng.*, 26(7):653–661, 2000.

[17] A. E. Hassan. *Mining Software Repositories to Assist Developers and Support Managers*. PhD thesis, University of Waterloo, Waterloo, ON, Canada, 2004.

[18] A. E. Hassan. Using development history sticky notes to understand software architecture. In *IWPC '04: Proc. of the 12th IEEE Int. Workshop on Program Comprehension*, page 183, Washington, DC, USA, 2004. IEEE Computer Society.

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

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

[21] P. Hsia. Making software development visible. *IEEE Softw.*, 13(3):23–26, 1996.

[22] P. M. Johnson, H. Kou, M. Paulding, Q. Zhang, A. Kagawa, and T. Yamashita. Improving software development management through software project telemetry. *IEEE Softw.*, 22(4):76–85, 2005.

[23] M. Jorgensen and K. Molokken-Ostvold. Reasons for software effort estimation error: Impact of respondent role, information collection approach, and data analysis method. *IEEE Trans. Softw. Eng.*, 30(12):993–1007, 2004.

[24] P. Kampstra. Beanplot: A boxplot alternative for visual comparison of distributions. *Journal of Statistical Software, Code Snippets*, 28(1):1–9, 10 2008.

[25] J. Kothari, A. Shokoufandeh, S. Mancoridis, and A. E. Hassan. Studying the evolution of software systems using change clusters. In *ICPC '06: Proc. of the 14th IEEE Int. Conference on Program Comprehension*, pages 46–55, Washington, DC, USA, 2006. IEEE Computer Society.

[26] 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: Proc. of the 4th Int. Symposium on Software Metrics*, page 20, Washington, DC, USA, 1997. IEEE Computer Society.

[27] M. Leszak, D. E. Perry, and D. Stoll. Classification and evaluation of defects in a project retrospective. *J. Syst. Softw.*, 61(3):173–187, 2002.

[28] A. Mockus, D. M. Weiss, and P. Zhang. Understanding and predicting effort in software projects. In *ICSE '03: Proc. of the 25th Int. Conference on Software Engineering*, pages 274–284, Washington, DC, USA, 2003. IEEE Computer Society.

[29] N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. In *ICSE '05: Proc. of the 27th international conference on Software engineering*, pages 284–292, New York, NY, USA, 2005. ACM.

[30] PostgreSQL official website. http://www.postgresql.org/.

[31] The Standish Group. The Chaos Report. Technical report, The Standish Group, 1994.

[32] M. van Genuchten. Why is software late? an empirical study of reasons for delay in software development. *IEEE Trans. Softw. Eng.*, 17(6):582–590, 1991.

[33] Z. Xing and E. Stroulia. Understanding phases and styles of object-oriented systems' evolution. In *ICSM '04: Proc. of the 20th IEEE Int. Conference on Software Maintenance*, pages 242–251, Washington, DC, USA, 2004. IEEE Computer Society.

[34] T.-J. Yu, V. Y. Shen, and H. E. Dunsmore. An analysis of several software defect models. *IEEE Trans. Softw. Eng.*, 14(9):1261–1270, 1988.