Studying Software Evolution Using the Time Dependence of Code Changes

by

Omar Alam

A thesis submitted to the
School of Computing
in conformity with the requirements for
the degree of Master of Science

Queen's University
Kingston, Ontario, Canada
May 2010

Abstract

Constructing software bears many similarities to constructing buildings. In a building project, each floor builds on the previous structures (walls of the previous floors) with some structures being more foundational (i.e. essential) for other structures and some periods of construction being more foundational (i.e. critical) to the final structure. In a software project, each change builds on the structures created by prior changes with some changes creating foundational structure and with some time periods of changes being more foundational than others.

This thesis is inspired by this similarity between constructing buildings and constructing software. The thesis makes use of the similarity to study the evolution of software projects. In particular, we develop the concept of time dependence between code changes to study software evolution through empirical studies on two large open source projects (PostgreSQL and FreeBSD) with more than 25 years of development history.

We show that time dependence can be used to study how changes build on prior changes and the impact of this building process on the quality of a project. We show how a development period impacts the development of future periods in a project. We also show how a subsystem (module) of a project builds on other subsystems and we identify the subsystems that have high impact on a project's development. Using

this knowledge, managers can better monitor the progress of the projects and better plan for future changes.

Related publications

The following are the related publications that are on the topic of this thesis:

- Omar Alam, Bram Adams, and Ahmed E. Hassan. Measuring the progress of projects using the time dependence of code changes. In ICSM '09: Proceedings of the 25th IEEE International Conference on Software Maintenance, pages 329-338, Edmonton, Canada, 2009. IEEE Computer Society. *(Acceptance ratio 21.6%,* ***Best Paper Award****)*

- Omar Alam, Bram Adams, and Ahmed E. Hassan. A study of the time dependence of code changes. In WCRE '09: Proceedings of the 16th Working Conference on Reverse Engineering, pages 21-30, Lille, France, 2009. IEEE Computer Society. *(Acceptance ratio 25.3%)*

- Omar Alam, Bram Adams, and Ahmed E. Hassan. Studying the Foundational Subsystems of Projects Using the Time Dependence of Code Changes. *Submitted to* ICSM '10: the 26th IEEE International Conference on Software Maintenance, Timisoara, Romania, 2010.

Authors Declaration for Electronic Submission of a Thesis:

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners. I understand that my thesis may be made electronically available to the public.

Acknowledgments

I would like to thank my supervisor, Prof. Ahmed E. Hassan for his academic support. Thanks Ahmed, for all the motivations, lessons, and advice. Special thanks to Dr. Bram Adams for his fruitful suggestions and discussions.

In addition, I appreciate the valuable feedback by my thesis readers: Prof. Jim Cordy and Prof. Thomas Dean.

I am fortunate to work in a great research environment of SAIL. I would like to thank the SAIL members for their encouragement and good treatment.

Finally, this thesis would not be possible without the continuous care from my parents and family, support of my friends, and guidance of my mentors. Their love and prayers are priceless to be recorded in words.

To my parents

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Measure what is measurable, and make measurable what is not so. -Galileo Galilei.

Many similarities exist between the construction of buildings and the evolution of software projects. A building is developed by laying a foundation that provides the major structure for developing the floors. Each floor provides the structure necessary to introduce the next floor. Similarly, a software project can be viewed as a building with changes building on the previous structure provided by changes. Some building projects spread horizontally by developing new foundations (such projects often lead to urban sprawls). Other buildings grow vertically by developing new floors (skyscrapers are an example of such projects). Similarly, software projects develop foundations that provide the basic structure for their development. As in buildings, we expect that some projects develop few foundational code (subsystems) while others develop considerable foundational code. Inspired by these similarities, this thesis proposes to study the evolution of a software project as the evolution of a building.

Software projects must continuously evolve to meet new requirements, to adapt

to a new environment or to fix defects, otherwise, the projects would become less satisfactory for the users [55]. Rich data in software repositories (like source control systems) motivated many researchers to mine this data to understand software evolution [7, 11, 31, 56, 65]. Tracking software evolution is usually done by measuring traditional metrics recovered from the repositories, like Lines Of Code (LOC), number of modules, number of changes, number of defects, or size of releases. Metrics that are based on counting LOC or number of changes provide a general understanding of the evolution of a project, but they do not allow to study how changes build on previous changes. For example, using counting metrics it is not possible to determine if the recently-added features are added to a project just-in-time or if they were added much earlier than needed. Counting metrics do not help to identify the subsystems of a project that are crucial for development of future subsystems. Therefore, new metrics are needed to study software evolution effectively. In this thesis, we formulate new metrics to study software evolution as the construction of a building.

In the past, researchers have often drawn analogies between buildings, and software architecture and software processes [69, 79, 94]. The World-Wide Institute of Software Architects [72] says:

"There is a compelling analogy between building and software construction. It is not new, but it has never taken root and bloomed"

We relate to the heart of software construction and development itself, for which concrete data is available in repositories. Therefore, the work in this thesis is able to empirically validate the use of time dependence, which is influenced by the building construction analogy, to study software evolution.

1.1 Research hypothesis

Our intuition leads us to formulate the following hypothesis:

The analogy between software projects and construction projects provides a different quantification of software evolution. This quantification provides valuable insight about the evolution of projects.

The goal of this thesis is to empirically explore this hypothesis through case studies on large and long-lived software projects.

1.2 Thesis overview

In this section, we give an overview of the first two parts of the thesis:

1.2.1 Part 1, Background

In this part, we discuss the related research to this thesis. We discuss the related research to building analogies in software and to software evolution. This part also discusses the concept of time dependence. Part 2 uses time dependence to empirically demonstrate our research hypothesis.

Related research

In Chapter 2, we first discuss the analogy of building construction and the past research that discussed this analogy. Then, we discuss the related research to software evolution and to the field of mining software repositories (MSR). Software evolution studies discuss implementing new features and fixing bugs. Therefore, we discuss

the related research to general software evolution and also to bug prediction using evolutionary data.

Time Dependence

We study the analogy of building construction in software by introducing the concept of time dependence between source code changes. Time dependence expresses temporal dependencies between "related" source code changes (like functions and variables) from which time gaps between the entities can be calculated. We use the time dependence to measure the progress and evolution of projects (for example, to know if the changes are added just-in-time or added much earlier than needed).

Getting accurate and timely feedback about the progress of a software project is critical to delivering high quality products on time and within budget [13, 22]. To get accurate feedback, 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 [80, 82]. Most of these failures are attributed to the discrepancy between the development process and project management [8, 41]. 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 also use time dependence to identify the foundational periods. Foundational periods are time periods that have high impact on the development of a project. As software builds 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 that build on it. Using this knowledge, managers can identify foundational periods in the lifetime of a project. Such foundational periods provide the structural foundation for a large number of future periods.

Managers can schedule extra testing and validation effort when source code from a foundational period is changed. Managers 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 can be consulted and possible retained, if needed, for better understanding and smooth development of future periods.

We further use time dependence to study the foundational subsystems (modules). Foundational subsystems are subsystems that have high impact on the development of other subsystems. Studying the foundational subsystems is important because software repositories continuously grow in size and become more complex over time [30, 49, 77]. The growth in the size of source code is usually accompanied with a growth of the development community, mailing lists and bug reports [83, 84]. Although more data makes decision-making more reliable, it becomes difficult for managers to have detailed understanding of all facets of their project. The increase in the size of a software project also results in the increase in the inter-dependence of the project's subsystems [51], which makes understanding the system even harder. Therefore, there is a need to identify important modules (foundational subsystems) of projects. Foundational subsystems provide more detailed knowledge than foundational periods. Good knowledge of the foundational subsystems can benefit project

managers. For example, managers can put extra effort on testing of such subsystems to ensure they are risk-free, provide training on these subsystems for the newly-hired people, ensure that the developers of these subsystems are around, and the documentation is well-kept.

1.2.2 Part 2, Empirical study

In this part, we validate our research hypothesis by conducting empirical studies on two large open source projects (PostgreSQL and FreeBSD) with over 25 years of development history. We apply the concept of time dependence at three levels. First, we study how changes build on prior changes. Second, we study how periods impact the development of future periods. Third, we study how subsystems build on other subsystems. We elaborate about each level in the following subsections:

How changes build on prior changes?

We measure the progress of a software project as the progress of a building project. Using time dependence, managers can determine if the changes are built as soon as the structure needed to introduce them is available or if they are delayed. New code changes typically build on prior changes. For example, a new function would build 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 entities that were added in prior changes. As with buildings, some changes can build on prior changes as soon as they were added (fresh structure) while other changes can build on well-established structure. Furthermore, some changes might build on no structure at all. As 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.

Our major empirical findings

- Over 50% of all code changes in a quarter build on newly-introduced changes. The ratio of changes that build on old changes, or that totally do not build on prior changes varies throughout the lifetime of a project and across projects. Large fluctuations in the rate are often associated with major structural changes and feature additions in the studied projects.

- Regular development and bug fixing activities are done on the same types of changes. Our finding could help project managers in allocating resources across both types of activities without worrying about the knowledge needed, because much of the knowledge is often shared across both types of activities (i.e., the context switch between both activities is relatively limited).

- Building on recent changes (fresh structure) is risky, as it leads to more efforts being spent on fixing bugs.

How do periods impact the development of other periods

After studying how changes build on prior changes, we study how a period impacts the development of future periods. We aggregate the changes in periods (e.g. quarters) and study how future periods build on the changes in a particular period. Understanding the time dependence between periods can help in better managing a project. For example, changes that are introduced in periods on which a large number of future

periods build (i.e., foundational periods) should be tested to be risk-free, the communication that took place on those periods (e.g., mailing lists) should be archived and studied, and the developers who were active in such periods must be retained and consulted. As a project stabilizes, we would expect that the number of foundational periods would drop. Indeed, most changes for such projects would be small and would only be incremental changes that build on previously established structure provided by past periods.

Our major empirical findings

- On average, up to 28% of the changes in a period build on changes made in the same development period.
- As a project ages, it tends to build less on changes from the current period, and more on older periods.
- Foundational periods (i.e. periods that have a large impact on future periods), are periods with huge restructurings, important new features or large imports of external source code.

How do subsystems build on other subsystems?

Here, we study the analogy of building construction at a higher level of abstraction. We study how subsystems build on other subsystems. While identifying the foundational periods helps to study how a period impacts the development on future periods in general, studying the foundational subsystems would allow to track the individual subsystems across periods (whether the periods are foundational or not). Managers

should put attention on testing the subsystems that have large impact on other subsystems. Managers also can ensure that the newly-hired people are trained about such subsystems and the documentation of these subsystems is up-to-date.

Our major empirical findings

- Projects develop few subsystems that provide their core structure. The highly foundational subsystems are core subsystems. The less foundational subsystems are either small in size or are introduced during inactive periods. The suddenly foundational subsystems are usually APIs or system libraries.
- Some subsystems have a large impact during limited periods, while other subsystems maintain their impact throughout their lifetime.
- The number of source code changes that are introduced in a subsystem is a good approximation of its impact on other subsystems. This makes it straightforward to determine the important parts of a project in practice, although this heuristic does not detect accurately all foundational subsystems.

Figure 1.1 summarizes the relation between changes, periods and subsystems. Projects consist of changes, which belong to subsystems. We apply time dependence on changes between different time periods (e.g. quarters or years).

1.3 Organization of thesis

This thesis is divided into three parts. The first part discusses the background and the metrics we developed to study our research hypothesis. The second part empirically explores our research hypothesis. The third part concludes our work.

Part 1- Background:

This part consists of two chapters. Chapter 2 discusses the related research to this thesis, and Chapter 3 discusses the the concept of time dependence. We use the concept of time dependence throughout the second part of this thesis to empirically study our research hypothesis.

Part 2- Empirical studies:

We start this part by discussing our data extraction and the case study set up in Chapter 4. In Chapter 5, we study how changes build on changes using time dependence (which can allow managers to study the progress of projects). In Chapter 6, we study how periods impact the development of other periods and study the foundational periods in the lifetime of a project. Practitioners can ensure that the communication (e.g. mailing lists) that took place in those periods is archived and well-kept. In Chapter 7, we study how subsystems build on other subsystems and study the foundational subsystems. Practitioners can ensure that the documentation of the foundational subsystems is up-to-date and the newly-hired people are trained about such subsystems.

Part 3- Conclusion: Chapter 9 concludes this thesis and outlines future work.

1.4 Major thesis contributions

In this thesis, we introduce the concept of time dependence between source code changes. We apply this concept on the changes retrieved from the source code repositories of projects. We then study how changes build on prior changes and the impact of this building process on the quality of a project. The summary of contributions of the thesis is as follows:

- We introduce the concept of time dependence, influenced by the building construction analogy, to study software evolution.
- We empirically demonstrate the benefit of using time dependence in studying software evolution. We perform case studies on two large and long-lived projects of more than 25 years of development history. We use the concept of time dependence:
  - to quantify the progress of projects by studying how changes build on prior changes. Applying the time dependence at the level of changes shows that building on newly added changes is risky as it leads to more bugs in the project.
  - to study how periods impact the development of future periods. As a project ages, it tends to build less on changes from the current period, and more on older periods.
  - to study how subsystems build on other subsystems. We show that the number of source code changes that are introduced to a subsystem is a good approximation of its impact.

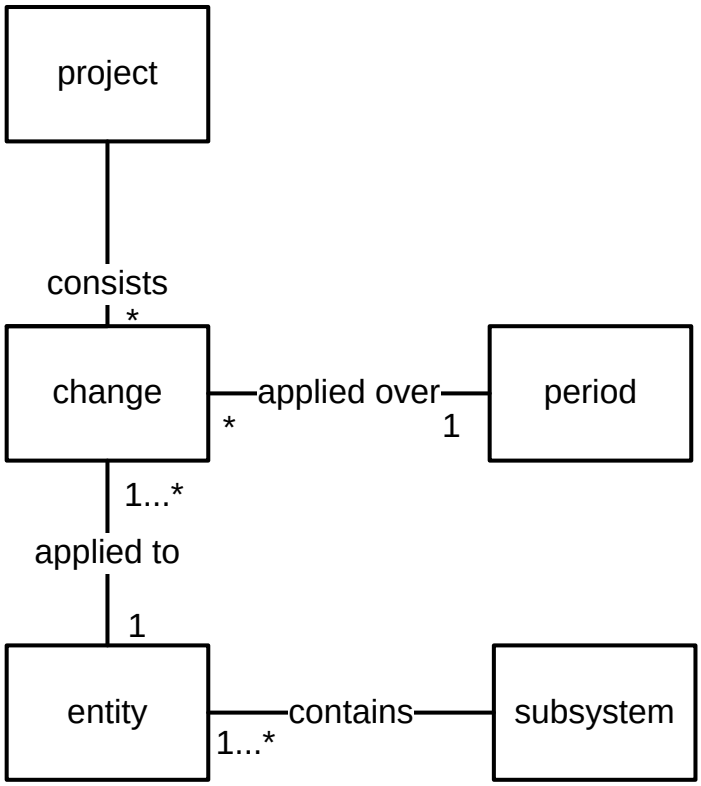


Figure 1.1: A figure showing the relations between changes, periods and subsystems. Projects consist of changes, which belong to different subsystems. The changes are aggregated in periods.

Part 1: Background

This part consists of two chapters. In Chapter 2, we discuss the related work to this thesis. There are two groups of related work that we discuss in Chapter 2: related work to the building analogies and related work to software evolution and mining software repositories studies. Chapter 3 presents the concept of time dependence. We explain the time dependence between changes, periods and subsystems. We use time dependence throughout Part 2 of this thesis to conduct our empirical case studies.

Chapter 2

Related research

In this chapter, we discuss the related research to this thesis. First, we discuss the related work to the building analogy in software projects and software project scheduling. We also discuss the related work to software evolution and to topics of mining software repositories (MSR). Software evolution studies the addition of new features to existing code or fixing defects that appear in the project. Therefore, in addition to general software evolution studies, we discuss the related work to bug prediction using evolutionary data.

2.1 Building analogies in software projects

There is a fair amount of published work that discusses the analogy between building construction and software [69, 79, 94]. These works draw similarities between a building construction process and software process. Construction engineers gather the requirements, plan the building, hire the builders, and buy the materials. Similarly, software project managers gather requirements, plan their projects, hire developers,

and buy licences and IDE's. The previous published work draws the similarity between these practices in software development without providing an empirical study of this analogy. This thesis is the first work to define the concept of time dependence to study software evolution, and to conduct an empirical study to validate this concept.

Some researchers discussed the importance of having views of the design before actual construction of software [52, 53, 69, 76]. These views are analogous to blueprints in building construction. Kruchten [52] proposed a "4 + 1" view model of software architecture. The "4 + 1" view model consists of the logical view, development view, process view, physical view, and scenarios. Each view is supported by a particular UML diagram. Clements et. al. and Hofmeister et. al. propose other view models of software architecture [18, 40].

Other researchers discussed the usefulness of design patterns [28, 78, 74]. Design patterns serve as templates for how to solve problems that appear in a large number of situations. Patterns have forms to describe the name, problem, context, forces, solution and sketch of the pattern [19]. Design patterns are mainly influenced by the work of a notable construction architect, Christopher Alexander, in pattern languages [5, 6].

A large part of research in project scheduling estimates the optimal project plan up-front (e.g., [1, 4, 45, 63]), 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, 63].

Research in software processes introduced process improvement models like the Personal Software Process (PSP) and Capability Maturity Model (CMM) [38, 42, 43]. The Personal Software Process has been developed to improve the process of small to medium size organizations and the Capability Maturity Model provides a maturity framework of five maturity levels for process improvement. Software process engineering also uses charts to describe dependencies and flow between project parts [17, 46, 57, 75]. Example charts are PERT charts, GANTT charts, CPM, and Petri Nets [17]. These charts identify the execution paths of the tasks, the duration of each task (start and end time), the name of the tasks, and the individual responsible for each task. This thesis extracts the progress feedback from the source code repository. Comparing the feedback produced from our approach with the planned charts can give managers understanding if the projects progress according to the plan or not.

Finally, there are efforts in 3D visualization to visualize software projects as cities and buildings [15, 48, 66, 67, 87, 88, 89]. Notably, the work of Wettel et al. visualizes classes as buildings and packages as tiles [88]. The height of the building is relevant to the number of the methods whereas the width of the building is relevant to the number of attributes. Visualizing an entire project results in a view of a city with buildings and parking lots. Our work is aimed to use the building construction analogy in measuring the progress and evolution of a software project. Our work quantifies the important parts of a software project by measuring how much a particular part impacts the future development of the project.

2.2 Research in software evolution and mining software repositories (MSR)

Research in software evolution [27, 30, 55] and software metrics [12, 44] detects or monitors development periods and areas with slow or rapid growth in a software project. However, these approaches examine the final outcome (the software system) instead of exploring the characteristics and temporal dependencies between changes.

Some researchers studied the evolution of similar code fragments of systems over different periods and releases. The work of Mende et. al. and Merlo et al. studied the similarity of code fragments over the releases of Linux kernel [58, 60]. They define similarity metrics to find code fragments that are similar to each other, but not necessarily identical. These works study clone relationships across releases instead of function call relationships as in our work.

There is also active research on the mining software repositories field (MSR). The MSR field analyzes the data in software repositories (e.g. source code repositories) to uncover actionable information about the software projects [32]. Our approach leverages information from historical data and builds the time dependence between changes to study the evolution of software projects. Mockus et al. [62] use historical data from version systems to identify code experts. Chen et al. [16] developed a tool called CVSSearch that uses the CVS comments to track source code fragments. Hassan and Holt [36] introduce the idea of attaching *Source Sticky Notes* to static dependency graphs, which assists in a better understanding of software architecture. Xing et al. [92] and Barry et al. [9] characterize the different periods of a software project using characteristics like spread of changes, effect of changes on dependency

structure and number of changes.

Other research studies 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 [61, 95]. 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.

Kothari et al. [50] 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 of "new development" encompasses all newly added development, whether or not it builds on earlier features or changes. Therefore, their approach cannot study how changes build on the previously-introduced changes.

The closest related work to this thesis is a proposal by Brudaru and Zeller to measure the genealogy of changes [14]. 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 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. Our approach harvests time dependence relations directly from the information stored in the source code repositories to keep the managers up-to-date.

German et al. propose the concept of a Change Impact Graph (CIG) to detect the impact of dependent changes when changing a source code entity [29]. 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 bugs. Our approach is not directly aimed at assisting developers during their daily development tasks. Rather, our approach provides managers in understanding which periods have provided the structural foundation on which later periods build.

Some researchers studied the evolution of software projects at the level of subsystems and plugins [30, 59, 86]. These studies investigate Lehman's laws [55] in open source projects by means of traditional metrics, like lines of code and size of files. These studies did not focus on the impact of the subsystems on the development of other subsystems.

Other researchers use complex network theory [64] to analyze software systems. They study the complex networks between software classes [64, 81], and packages [85, 54], and find that the distance between the nodes is usually small (small-world behavior). The implementation by Wen et. al. [85] can also identify important components from the component dependency network. However, these studies are performed on the source code without using historical information from repositories. Therefore, these approaches are not able to identify important components at different time periods.

Software evolution discusses fixing bugs in addition to implementing features. Therefore, in this thesis, we study how building on new changes is associated with bugs. Although some approaches detect bugs based on metrics like LOC [31, 39, 56] or on the presence of prior faults [93], the majority of recent techniques are based on information from the change history of a system, as extracted from the source code repository [7, 11, 31, 56, 65]. These techniques typically look at code churn [65] or the number of changes. For example, Bernstein et al. [11] 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 that use other types of historical data for predicting bugs.

2.3 Chapter summary

In this chapter, we discussed the related work to this thesis. We first discussed work related to building analogies. We found that the previously published work that discusses the building analogy does not provide any empirical evidence to demonstrate the benefit or support such analogy. We also discussed the related work to project scheduling. Our approach is unique in a sense that it extracts the feedback about the progress of projects from the code changes in the software repositories and not from progress reports. Then, we discussed the work related to software evolution in general and other related work to defect analysis using evolutionary data.

Chapter 3

Time dependence

Traditional software evolution studies do not study the interaction between different time periods [27, 30, 55, 96], i.e., they do not study how much a version builds on the previous versions. Time dependence can be used to calculate the time gap between "related" source code changes between different time periods with each changed code entity is located in a subsystem of a project. Therefore, time dependence is aware of "space" and "time" where source code changes happen. Figure 3.1 illustrates the awareness of space and time. Change 2, which belongs to subsystem 2 and is changed at time 2, builds on (time dependent on) the previously introduced change 1, which belongs to subsystem 1 and was introduced at time 1. Time dependence allows us to relate changes between version(s) and/or subsystem(s) throughout the lifetime of a software project.

We use time dependence to study the development of a software project at three levels:

- **Level 1: Time dependence between changes**, which allows us to study

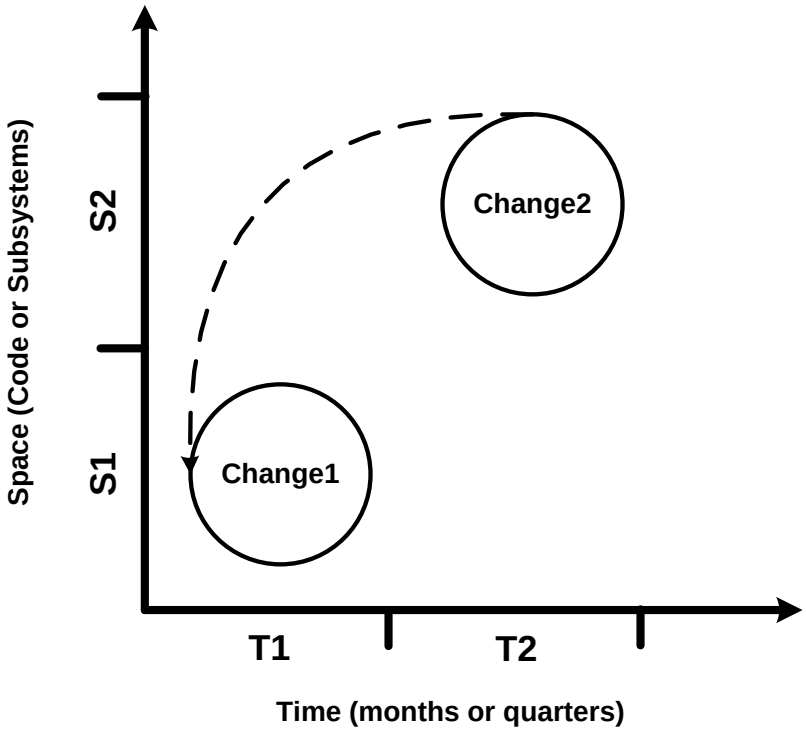


Figure 3.1: The relation between space and time between code changes. Time dependence allows to relate Change 2 that belongs to Subsystem 2 at Time 2 to Change 1 of Subsystem 1 at Time 1.

the progress and evolution of projects.

- **Level 2: Time dependence between periods**, which allows us to study the foundational periods. Studying foundational periods would help practitioners to identify the communication (e.g. mailing lists) that took place on such periods.

- **Level 3: Time dependence between subsystems**, which allows us to study the foundational subsystems. Studying the foundational subsystems would allow managers to ensure that the documentation of such subsystems is up-to-date and the newly hired people are trained about such subsystems.

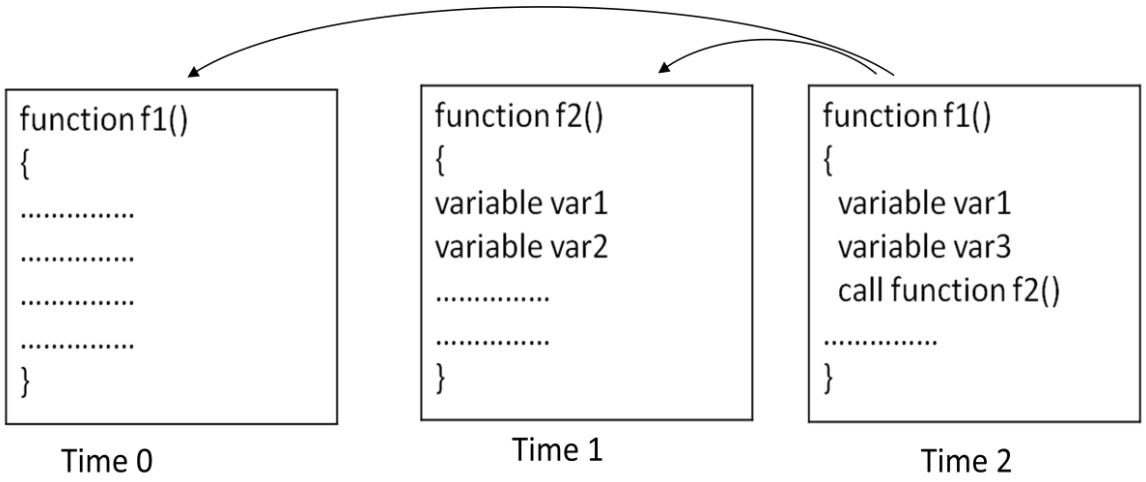


Figure 3.2: Time dependence between changes.

3.1 Time dependence between changes

The problem of measuring the evolution (or 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 right now. 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) [30, 55], we propose the concept of time dependence to quantify our intuition about the progress of a project.

Each change can build on a large number of prior changes, but for this level we consider the smallest time interval between a change and any of its dependent

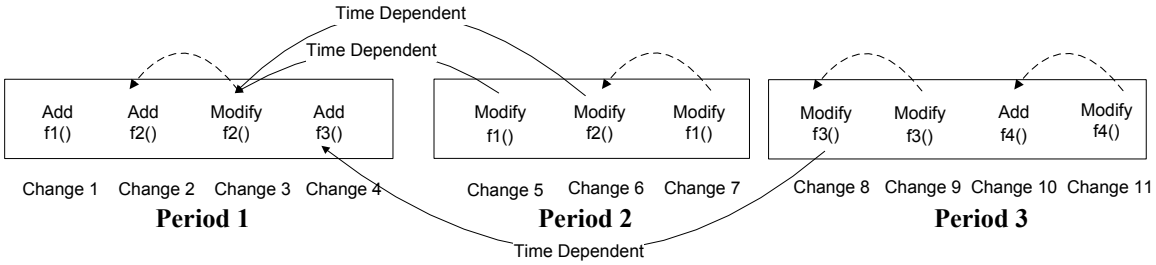


Figure 3.3: Time dependence between periods before resolving transitive dependence.

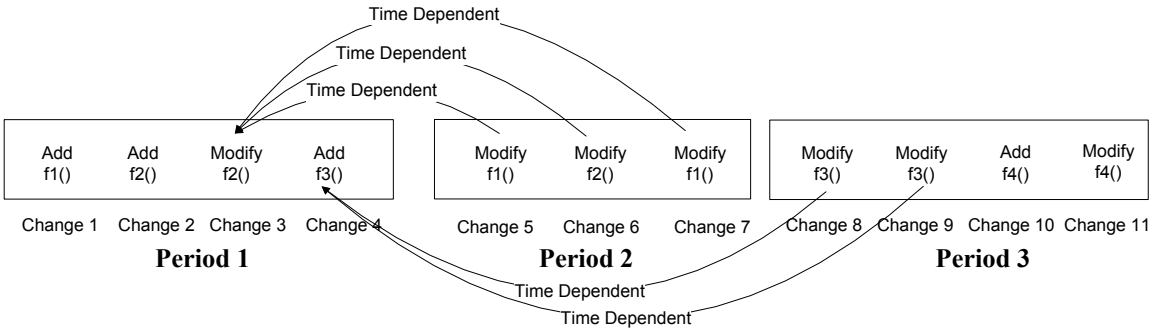


Figure 3.4: Time dependence between periods after resolving transitive dependence.

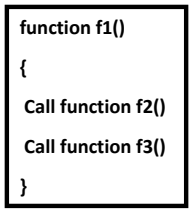
changes. In Figure 3.2, function ***f1*** is added at *time 0*. Function ***f2*** is added at *time 1*. At *time 2*, ***f1*** is modified to add a call to ***f2***. We would say that the change to ***f1*** at *time 2* depends on the change at *time 1*. This indicates that the earliest possible time to 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.

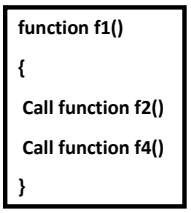
To provide a high-level view of the progress of a project, we study all changes in a particular period (e.g., a month, a quarter, or a year). Whatever definition of period we use, we must "lift" all time dependence relations within a particular period. Figure 3.3 and Figure 3.4 illustrate the lifting process. Before the lifting process, three periods are shown on Figure 3.3, 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 in Figure 3.4, 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 period.
- **independent change** does not depend on changes in any prior period.

Figure 3.4 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.



(a) initial version of the function before applying the changes



(b) the function after the changes are applied

Figure 3.5: Function f1() before Figure 3.5(a) and after (Figure 3.5(b)) making changes.

3.2 Time dependence between periods

In this level, we use the time dependence relation to link 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) to make sure there is a time dependence relation corresponding to added *and* removed function calls.

We introduce the following definitions at this level:

- **time dependence between periods:** In order to study time dependence across the entire history of a system, we lift up the time dependence of individual changes between two specific time instants to ***time dependence of individual changes between two periods*** such as quarters or years.

- **within-period time dependencies:** Within-period dependencies happen when an edge starts and ends in the same period, i.e., self-loops. These indicate that a period builds on itself.

- **Outer-dependencies:** are when an edge starts in a period and ends in a

different period.

- **backward time dependence of a period:** The ***backward time dependence of a period*** produces the set of outgoing time dependence edges that point to all prior periods on which the current period builds. The larger the number of edges produced when calculating the backward time dependency in a period, the more this period builds on prior periods.

- **forward time dependence of a period:** The ***forward time dependence of a period*** produces the set of incoming time dependence edges that come from all future periods that build on this current period. The larger the number of edges produced when calculating the forward time dependency, the more foundational a period is.

- **foundational periods:** Foundational periods are periods on which a large number of other periods build (i.e., periods that have large numbers of forward time dependence relations). We expect that in practice systems have ***foundational periods*** on which new changes continuously build. These periods contribute essential code that forms the structural foundation on which future changes build. An example of such structural foundation are changes that 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.

Figure 3.6 describes the time dependence between periods in a small example.

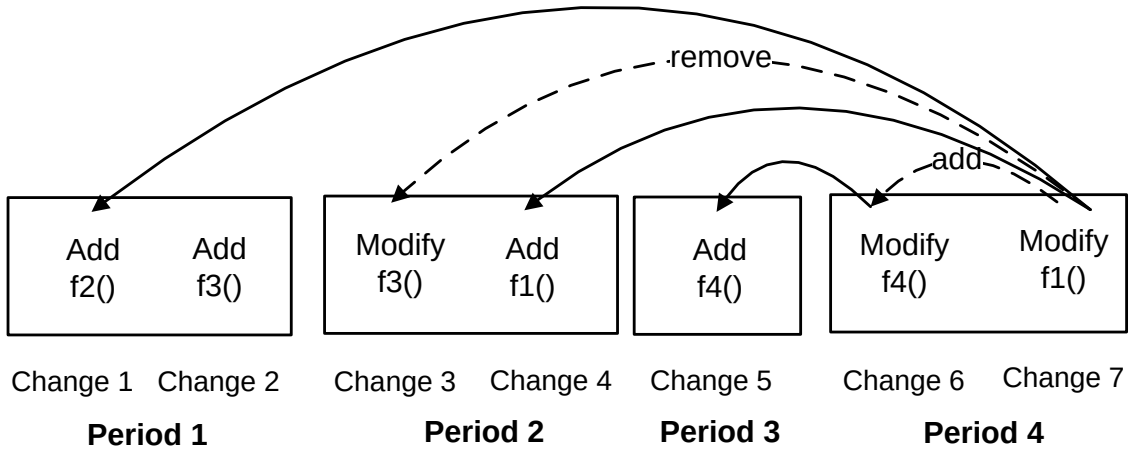


Figure 3.6: Time dependence relations for the example system before (Figure 3.5(a)) and after (Figure 3.5(b)) making changes. The arrows in Figure 3.6 connect changes 6 and 7 to all changes they build on. The dashed arrow connects a change to the most recent change of any entity to which it added or removed a reference (e.g. function call).

Changes happen in four time periods. A developer modifies function ***f1()*** (Figure 3.5(a)) in change 7 by removing the call to ***f3()*** and adding a call to ***f4()*** (Figure 3.5(b)). Change 7 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. Rectangular nodes represent periods in Figure 3.6 and the edges express the time dependence relations between changes in these periods. We calculate the ***age of a time dependence relation*** as 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 3.6 is two periods.

Figure 3.6 only shows edges for changes 6 and 7, but similar edges can be drawn for other changes as well. We calculate the forward and backward time dependence of changes in each period. For example, in period 2, change 1 and change 3 have one forward time dependence each, while change 1 has five backward time dependence in

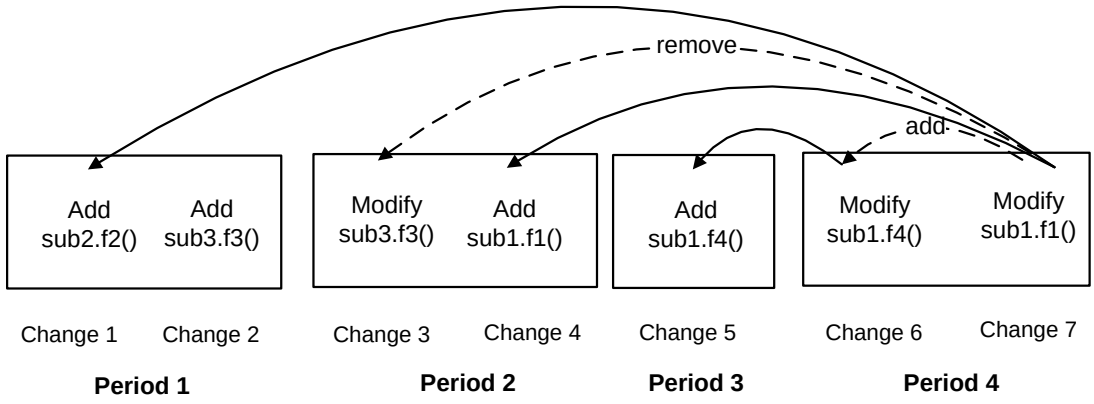


Figure 3.7: Time dependence relations for the example in Figure 3.6 with each change related to its corresponding subsystem.

period 4. Similar forward and backward time dependence relations are drawn for the entire periods as well.

3.3 Time dependence between subsystems

As individual changes might be too fine-grained, we further lift up the time dependence of individual changes between periods to ***time dependence between subsystems***. Hence, time dependence has a notion of space (subsystems) and time (periods). Here are the definitions that we introduce at this level:

- **a backward time dependence of a subsystem:** is an outgoing time dependence edge from a subsystem at a given period to a subsystem at a previous period. The larger the number of edges produced when calculating the backward time dependency of a subsystem, the more this subsystem builds on subsystems in older periods.

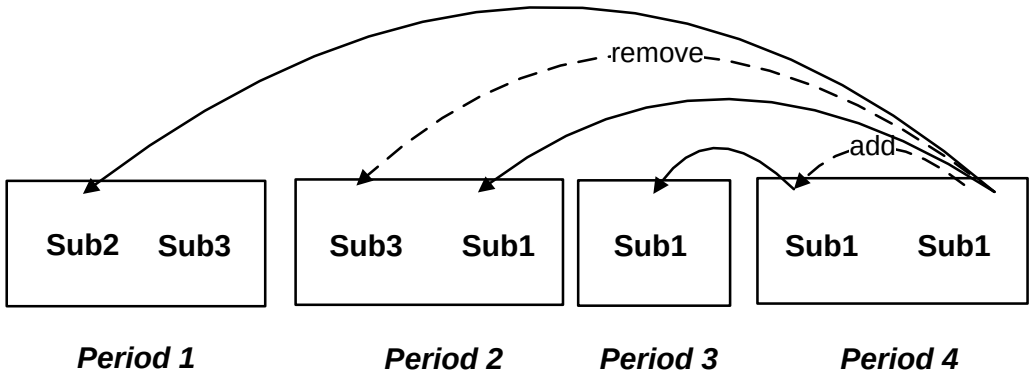


Figure 3.8: Lifting up the changes in Figure 3.7 to subsystems and applying the time dependence relations between them.

- **a forward time dependence of a subsystem:** is an incoming time dependence edge to the subsystem from a future subsystem. The larger the number of edges produced when calculating the forward time dependency of a subsystem, the more foundational the subsystem is.

- **foundationality** [1] **of a subsystem:** in a particular development period corresponds to the number of forward time dependence relations a subsystem has at that given period. The more forward time dependence relations a subsystem has, the higher impact it has on the future development of the project. Subsystem A is more foundational than subsystem B if subsystem A has more total forward time dependence relations during its lifetime than subsystem B. Studying foundational subsystems would allow managers to ensure that the documentation of such subsystems is up-to-date and newly hired people are trained

[1] Although the word foundationality does not exist in the English dictionary, it has been used by philosophers and even by some computer scientists as in [20].

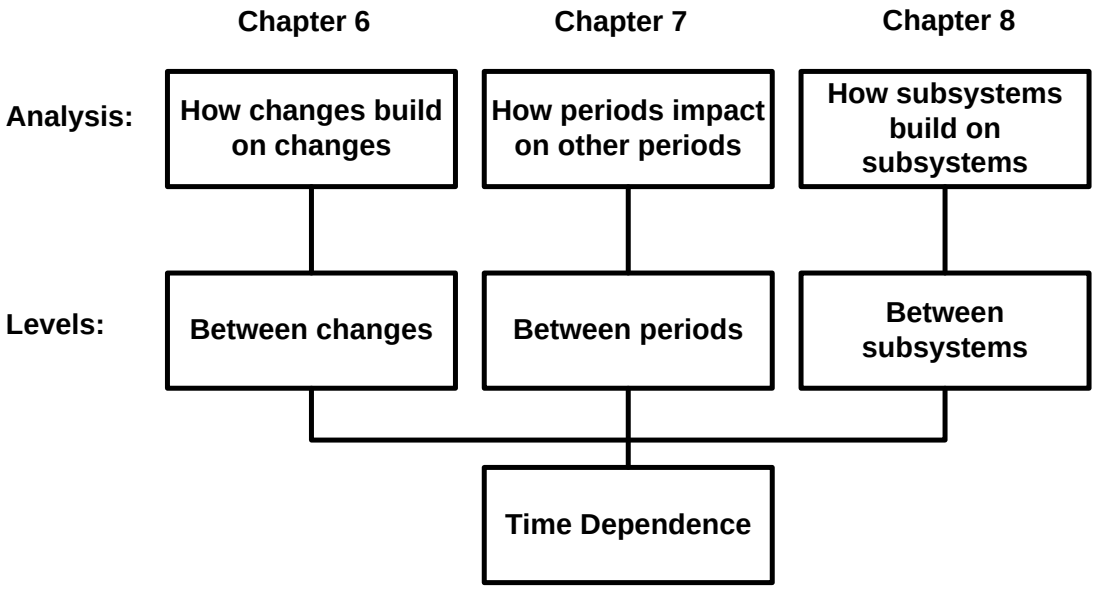


Figure 3.9: A figure summarizing the different levels of time dependence relations.

about such subsystems.

As an example, the entities that are changed in Figure 3.6 are annotated with the subsystems they belong to in Figure 3.7. We lift up the changes to subsystems and establish the time dependence between subsystems in Figure 3.8. In Figure 3.8, subsystems sub 1 and sub 3 have one forward time dependence each in period 2, while sub 1 has five backward time dependence in period 4.

We expect that in practice systems have foundational subsystems that provide the structure for their future development. Foundational subsystems can be APIs or system libraries that provide the essential structure for development of subsystems in future periods.

3.4 Chapter conclusion

In this chapter, we discussed the concept of time dependence between source code changes. As illustrated in Figure 3.9, we use time dependence to study how changes build on changes, to study how periods impact the development of other periods, and to study how subsystems build on subsystems. For studying the progress of projects, we establish time dependence between a change and the most recent change of all the changes that it builds on. For studying how periods impact other periods, we establish the time dependence relation between a change and the last change of each source code entity in its call graph. For studying how subsystems build on other subsystems, we relate each change to the subsystem it belongs to.

Part 2: Empirical studies

In this part, we discuss the empirical studies that we conducted to validate our hypothesis. This part consists of four chapters. Chapter 4 provides a glossary of the terminology used in this part. In Chapter 5, we discuss the data extraction and experimental setup. In Chapter 6, we study how changes build on changes. In Chapter 7, we study how periods impact the development of future periods. In Chapter 8, we study how subsystems build on subsystems.

Chapter 4

Glossary for Part 2

For the convenience of the reader, this chapter summarizes the terminology that we use in our empirical studies. We provide the summary definitions of the terminology at each level of time dependence from Chapter 3, and we start with time dependence between changes.

4.1 Time dependence between changes

This is the first level of the time dependence relation that we study. As discussed in Section 3.1, we establish a time dependence relation between a change and the **most recent change** it builds on. We use the following terminology at this level:

- **entity:** a source code entity like a function, variable, type definition, etc.
- **change:** a change to a source code entity (i.e., insertion, modification or deletion of an entity).

- **time dependence:** Time dependence expresses temporal dependencies between "related" source code changes (like functions and variables) from which time gaps between the entities can be calculated.
- **time dependence between changes:** links a change of a source code entity E (like a function or type definition) at time T, to the last change of E and of each entity that E builds on.
- **built-on-new change:** builds on a change in a recent period.
- **built-on-old change:** builds on a change in an older period.
- **independent change:** does not build on changes in any prior period.
- **bug fixing change:** introduced to fix a bug.
- **enhancement change:** introduced for regular development (non-bug fixing change).

4.2 Time dependence between periods

Here we discuss the terminology of the second level of time dependence. We establish the time dependence relation between periods, as discussed in Section 3.2. We use the following terminology at this level:

- **period:** time period like month, quarter and year. We mostly use quarters and years in this thesis.
- **time dependence between periods:** time dependence relation that links a change of a source code entity E (e.g., a function or type definition) at period

P to the most recent change before P of E and of *each entity* that E depends on before or after the change (via its call graph).

- **within-period time dependence:** time dependence edge that starts and ends in the same period, i.e., self-loops.
- **outer-period time dependence:** is a time dependence edge that starts in a period and ends in a different period.
- **backward time dependence of a period:** all outgoing time dependence edges from a period. These edges point to all prior periods on which this current period builds.
- **forward time dependence of a period:** all incoming time dependence edges at a period that come from future periods.
- **age of a time dependence relation:** is the time difference between the source and destination periods of a time dependence relation.
- **foundational periods:** are periods that have a large number of forward time dependence.

4.3 Time dependence between subsystems

Finally, we discuss the terminology of the third level of time dependence. Here, we study the time dependence between subsystems. We use the following terminology at this level, introduced Section 3.3:

- **subsystem:** this thesis considers the second level directories as subsystems of the PostgreSQL project(e.g. odbc in /interfaces/odbc/Attic/) and fourth top level directory as subsystems of the FreeBSD project (e.g. dev in /freebsd/src/sys/dev/cxgb/), similar to the approach by Godfrey et. al. [30].

- **time dependence of subsystems between subsystems:** time dependence that links a subsystem A at period P to the most recent changed entities in A and the recent changes of all other subsystems B that entities inside A build upon.

- **backward time dependence of a subsystem:** all outgoing time dependence edges from a subsystem at a given period to those development periods of other subsystems it builds on.

- **forward time dependence of a subsystem:** all incoming time dependence edges at a subsystem that come from subsystems from the future periods.

- **foundationality of subsystem:** subsystem A is more foundational than subsystem B if subsystem A has more total forward time dependence relations during its lifetime than subsystem B.

- **foundational subsystems:** subsystems that have high forward time dependence relative to all other subsystems in the project. We usually identify the top 10 or top 20 foundational subsystems relative to all other subsystems in the project.

This concludes the summary of terminology that we use in this thesis. Chapter 3 provides detailed definitions of this terminology.

Chapter 5

Setup for case studies

In this chapter, we discuss our case study setup and how we extract the data from the CVS repositories of the projects. We conducted several case studies on two large open source projects, PostgreSQL and FreeBSD, of more than 25 years of development history. We give brief details about the studied projects and our data extraction process. We discuss how old the studied projects are, and what their characteristics are.

5.1 Studied systems

For our case study, we used data from the open source PostgreSQL (1996–2007) and FreeBSD (1993–2009) projects. PostgreSQL is a relational database system of which the original design goes back to the 1980s [73], whereas FreeBSD is an operating system distribution derived from the Berkeley flavor of UNIX [24]. We studied the time dependence at the level of quarters, since it is a common time period for project planning [35] (other time periods could be explored using our approach). We picked

	PostgreSQL	FreeBSD
type	DBMS	Operating System
studied period	1996–2007	1993–2009
number of changes	84,311	1,074,858
number of entities	31,863	617,000
number of files	2,053	37,724
number of bugfixes	22,913	144,582
number of subsystems	65	958

Table 5.1: Characteristics of the studied systems.

both systems due to their long and archived history of changes, as Table 5.1 shows. The two systems being from two different domains (databases and operating systems) would help us validate the generality of our findings across domains.

5.2 Data extraction

We believe that the required information for evaluating the evolution 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. However, these repositories contain data at the level of lines of codes (LOC). To lift the line-level information of changes to the level of source code entities like function calls and type definitions, we use a technique developed by Hassan and Holt [36]. By decorating the time dependence relations with the metadata attached to them (like commit messages and developer names), we are able to figure out the prior changes on which a particular code change builds. Using the metadata, we can distinguish between ***bug fix changes*** and other, non-bug fixing ***enhancement***

changes. 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 using a lexical approach developed by Hassan [34].

The approaches to lift the data from LOC level to the entity level [36], and to identify the types of changes [34] (e.g., bug fix or enhancement) are used to develop an evolutionary code extractor for C-based systems called C-REX [33]. We use C-REX to extract the change history from the Concurrent Versions System (CVS). C-REX tracks the addition, modification and deletion of source code entities and records them as groups of related changes (changelists). A changelist consists of a group of changes that a developer introduced to implement a feature. The C-REX output data is in XML format and consists of the following parts:

- Historical Symbol Table: records all the entities that ever existed in a project.
- Changelists: each element of a changelist records a group of related changes. The changes are recorded as child elements of the changelist.

A changelist element has attributes that indicate the number of files changed, the name of the developer who introduced the changelist, the commit time of the changelist to the source code repository, a unique hash code to identify the changelist, and keywords of frequent words appearing in the changelist. The changes within a changelist are recorded as child elements. The name of the change element reflects the change type (i.e. removal, addition, or modification of the entity). Each change element has attributes to describe the name of the entity, keywords that appeared in the source code comments, normal string keywords, dependency (call graph) keywords, file name, and the revision number.

```
1. <CHANGELIST_DETAILS FILE_COUNT=``8'' TIME=``870853301'' HASH=``2081
   e58de1aaf6f04aa8cffc6416415d'' AUTHOR=``momjian'' TYPE=``BUG''>
2.  <ADD_ENT NAME=``inittapes'' .../>
3.     :
4.  </ADD_ENT .../>
5.     :
6.   <MODIFY_ENT NAME=``mergeruns'' TYPE=``function'' DEPENDENCY_KEYWORDS=``assert
   3, sort -2'' FILE_NAME=``./backend/.../psort.c'' REV_NUMBER=``1.6''>
7.     <ModifyTime Type=``Entity''>836907737</ModifyTime>
8.     <ModifyTime Name=``assert'' EntityTpe=``function'' FileName=``./backend/.../
   psort.c''>846763813</ModifyTime>
9.     <ModifyTime Name=``sort'' EntityTpe=``function'' FileName=``./backend/.../
   psort.c''>870853301</ModifyTime>
10.    <ModifyTime Name=``level'' EntityTpe=``variable'' FileName=``./backend/.../
   psort.c''>836907737</ModifyTime>
11.    <ModifyTime Name=``level'' EntityTpe=``variable'' FileName=``./backend/.../
   psort.c''>870853301</ModifyTime>
12.    <ModifyTime Name=``taperange'' EntityTpe=``variable'' FileName=``./backend
   /.../psort.c''>870853301</ModifyTime>
13.  </MODIFY_ENT>
14. <CHANGELIST_DETAILS/>
```

Listing 5.1: Example of C-REX output

Listing 4.1 shows an example of a changelist in C-REX. The `DEPENDENCY_KEYWORDS` attribute lists the entities that are added or deleted from the call graph of the function `mergeruns`. The number following each entity indicates how many instances of the entity has been added or deleted. The `assert` entity in `DEPENDENCY_KEYWORDS` was added three times and two instances of `sort` were deleted.

Our approach processes the C-REX data and records each change in a hash table.

Each entry in the hash table contains the entity name, and the added and deleted entities from the entity's call graph. For example, if the `assert` entity already exists in the `mergeruns` hash entry, number 3 will be added to its occurrence count in `mergeruns`. Similarly, number 2 will be subtracted from the occurrence number of the `sort` entity in the `mergeruns` hash entry. If the result after subtraction of `sort` is 0, the `sort` entity will be totally deleted.

We note that C-REX does not record the types or the file locations of entities in `DEPENDENCY_KEYWORDS`. Therefore, we use heuristics to select between multiple entities with the same name. We first look for the `DEPENDENCY_KEYWORDS` entity in the same file of its calling entity (`mergeruns` in our example), if the entity name does not exist in the same file, we look for the entity in the same subdirectory of the calling entity, if it does not exist in the subdirectory, we look in the parent subdirectory, and so on. After we locate an entity from `DEPENDENCY_KEYWORDS`, we attach the located entity's past modification times from the hash table as subelements of the change.

Listing 4.1 shows modification times of the `mergeruns` and its called entities. We first look in the hash table for the modification times of `mergeruns` and list them (`mergeruns` was changed only once in the past). We then record the modification times for the entities that are added or deleted from the call graph of `mergeruns` (i.e., entities in `DEPENDENCY_KEYWORDS`). After that, we record the modification times of entities that already exist in the call graph of `mergeruns` (which are not in the `DEPENDENCY_KEYWORDS`). The first modification time for `mergeruns` in Listing 4.1 is for the entity itself, which was changed at UNIX timestamp 836907737 (a UNIX timestamp counting seconds since 1970). Note that we record all change times for an entity. For example, the entity `level` was changed twice (in line 10 and 11 in

Listing 4.1) before the change to `mergeruns` was introduced. We further process the modification times to study how changes build on changes, how periods build on periods, and how subsystems build on subsystems. For time dependence between changes (discussed in Section 3.1), we take the minimum of all modification times of an entity and its called entities (i.e., 870853301 for `mergeruns`). For time dependence between periods (discussed in Section 3.2), we need to take the recent modification times for the calling entity and for each called entity. We then apply the time dependence on snapshots of periods. For time dependence between subsystems, we lift the time dependence to the subsystem level. This thesis considers the second level directories as subsystems of the PostgreSQL project (e.g., odbc in /interfaces/odbc/Attic/) and fourth top level directory as subsystems of the FreeBSD project (e.g. dev in /freebsd/src/sys/dev/cxgb/), similar to the approach by Godfrey et. al. [30].

5.3 Chapter conclusion

In this chapter, we discussed the studied projects in this thesis. We studied two large open source projects, PostgreSQL and FreeBSD. These two projects come from different domains, which allows us to generalize our findings (PostgreSQL is a database management system and FreeBSD is an operating system). The development data for the projects is stored in repositories like Concurrent Versions Systems (CVS). We used C-REX (an evolutionary code extractor for C-based system) to mine the data from the CVS repositories of the two projects. We extracted our different time dependence relations from the output of C-REX.

Chapter 6

How do changes build on changes?

In this chapter, we discuss how we use the concept of time dependence to measure the progress and evolution of projects. An earlier version of this chapter was published in [2]. Using the approach discussed in Section 3.1, we answer four research questions, the first two questions are related to the nature of tracking the progress of projects and the other two address two common project management beliefs.

6.1 How does the time dependence of *changes* vary over time?

This question discusses how code changes build on newly introduced code, old and stable code, or do not build on any existing code. We study the distribution of these types of building over the lifetime of a project. We study if 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. Using our dependence analysis and their

knowledge of a project, managers can monitor more closely the progress of a project.

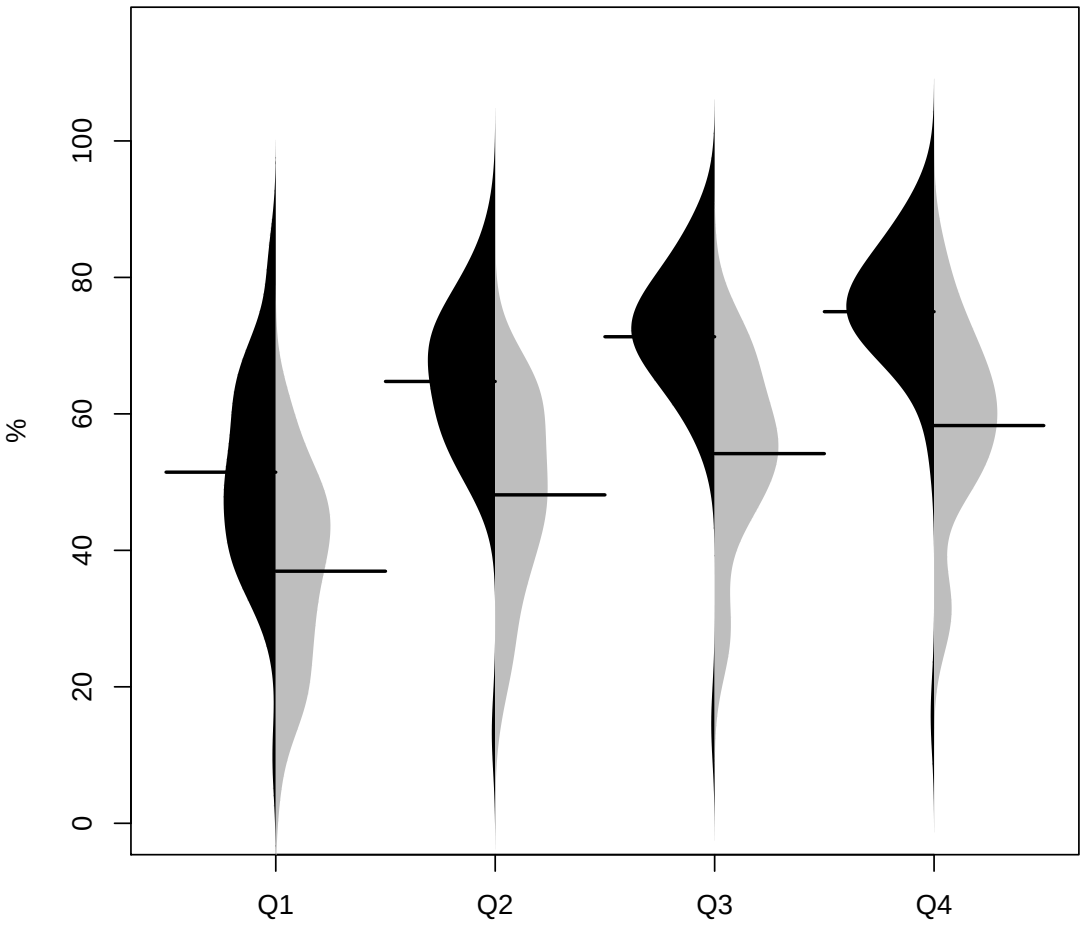


Figure 6.1: Beanplots with the percentage of changes built on last quarter, last two quarters, last three quarters, and last four quarters. Each beanplot compares the data for PostgreSQL (left) and FreeBSD (right).

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,

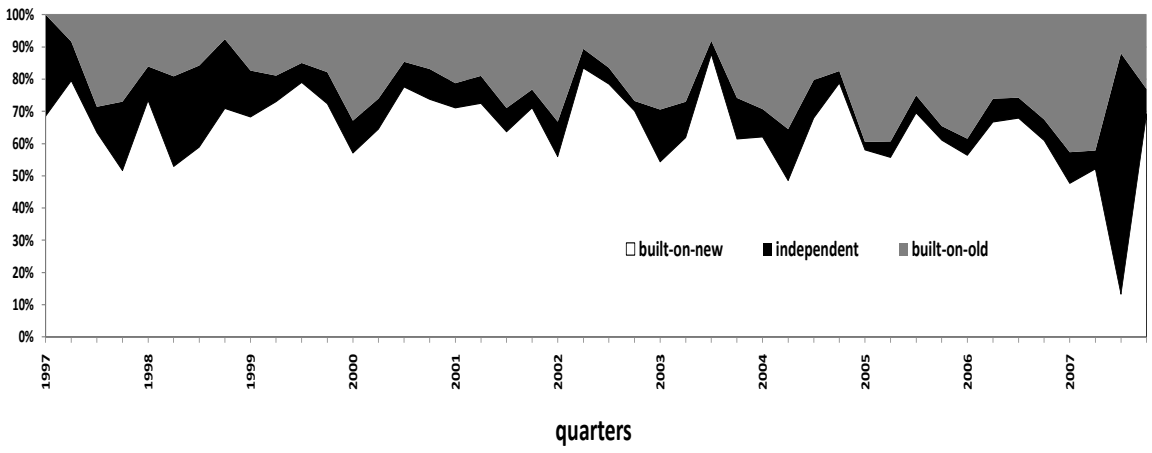


Figure 6.2: The distribution of built-on-new, built-on-old and independent changes (the PostgreSQL project).

three, and four quarters. These percentages are represented in Figure 6.1 as four beanplots [47], 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). Beanplots facilitate 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 when looking at time dependence relations of age higher than two quarters (see Figure 6.1). For PostgreSQL, on average 51.2% of the changes in a quarter depend on the last quarter, whereas on average 64.8% depend on the last two quarters. For FreeBSD, the averages are 47.3% and 57.7%, respectively (Q1 and Q2 in Figure 6.1). When moving to periods longer than two quarters (e.g., from two to three quarters), we only gain a small increase, less than 10% (Q2 and Q3 in

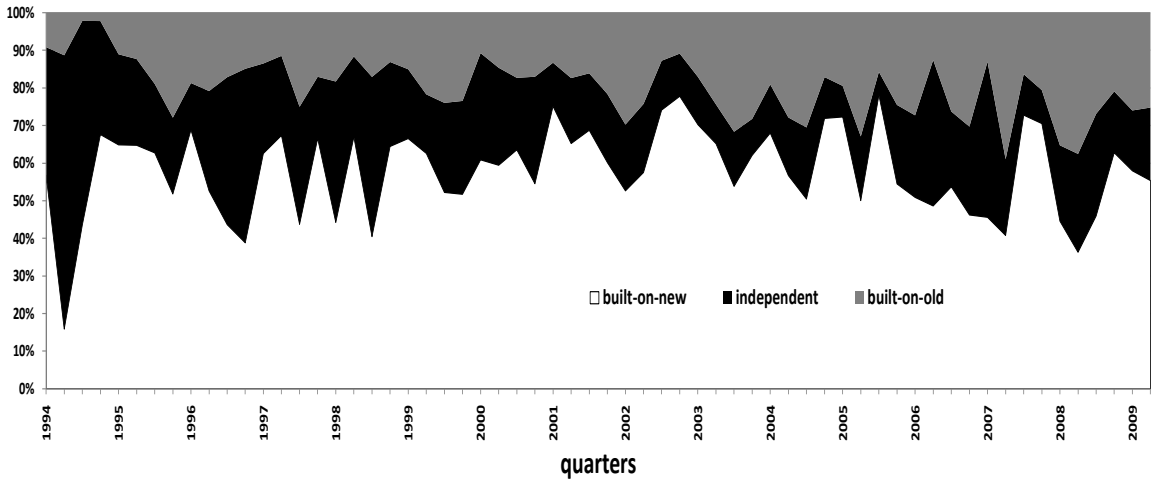


Figure 6.3: The distribution of built-on-new, built-on-old and independent changes (the FreeBSD project).

Figure 6.1). 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 6.2 and Figure 6.3 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 and FreeBSD.

There is a large number of independent changes in FreeBSD over time, possibly due to architectural characteristics of FreeBSD over PostgreSQL. The large number of independent changes 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 base for PostgreSQL-related tools has a much smaller scale. There are only a few periods for PostgreSQL (2007) and for FreeBSD (1994) where the percentage of built-on-new changes drops below 20%. Those periods saw an unusually high number of independent changes (we explore these unusually higher number of independent changes in the following section).

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 that we found. The next section 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. Addition of built-on-new changes 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 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. The FreeBSD project has more independent changes than the PostgreSQL project. Both projects build more frequently on newer changes.

6.2 What is the impact of independent changes?

Independent changes are "floating" changes that 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 6.4 plots the black areas of Figure 6.2 and Figure 6.3 on one graph.

Figure 6.4 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

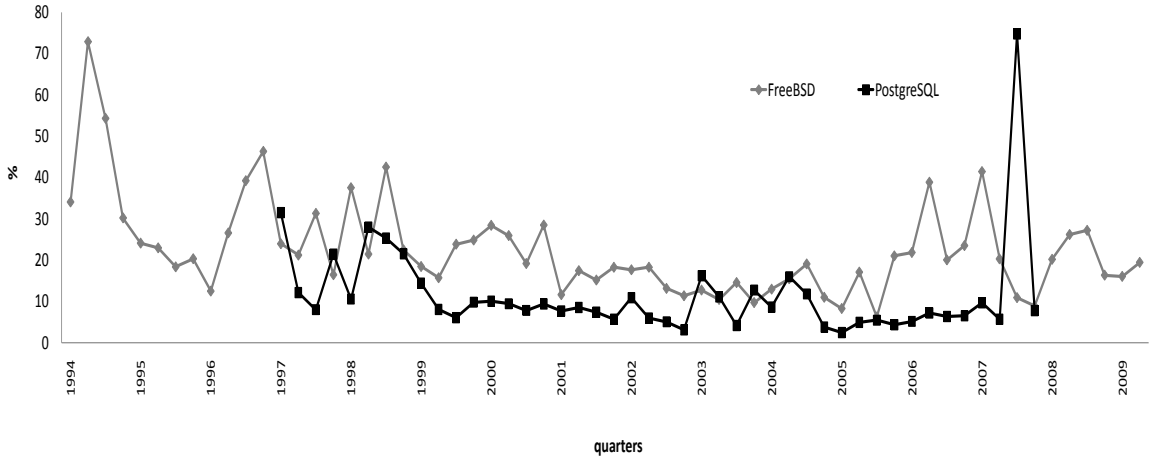


Figure 6.4: Percentage of independent changes in PostgreSQL and FreeBSD.

experienced a large spike in the percentage of independent changes. We validated this finding by contacting the PostgreSQL developers. These explained that two large independent features had been added to the backend in August and September 2007, in preparation of the 8.3 release [21]:

- Tom Lane, a major PostgreSQL developer, migrated the *Tsearch2* functionality ("Text Searching") from the separate "contrib" 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 continuously imports and integrates large chunks of external source code, whereas spikes of independent changes in PostgreSQL signal sudden additions of large independent features.

6.3 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

	PostgreSQL	FreeBSD
year	1996-2007	1993-2009
built-on-new	0.75	0.80
built-on-old	0.89	0.72
independent	0.62	0.78

Table 6.1: 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 ($p\text{-}value < 0.05$).

build on similar change periods or do they differ? For example, are there periods where bug fixing depends 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 developer productivity due to the interleaving of bug fixing with regular development [82]. In other words, a developer can fix bugs using the same mindset for regular development (i.e., no context switch is required).

Using our methodology of time dependence in Section 3.1, we can boil down this problem 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 (see Section 3.4). We measure the correlation between built-on-new enhancement changes and built-on-new bug fixing changes in year basis, and similarly, we calculate the correlation between enhancement and bug fixing changes for built-on-old and independent changes. Figure 6.5 shows the types of changes we are correlating. 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 6.5. These correlations will show us if 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 if such periods have a small percentage of built-on-new bug fixes, leading to a negative correlation.

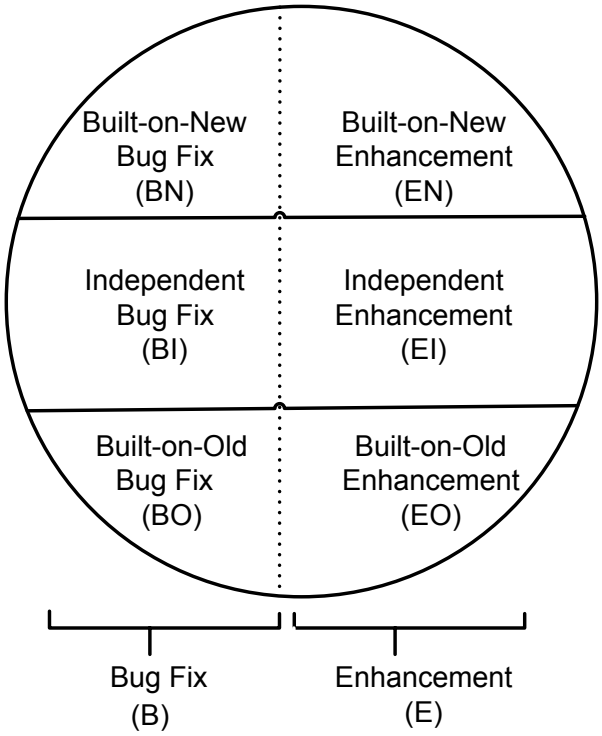


Figure 6.5: Categories of changes.

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 6.1). 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 Section 5.1, we use one year as the boundary between built-on-new and built-on-old changes.

PostgreSQL and FreeBSD in Table 6.1 obtain strong correlations for the distribution of time dependence for enhancement and bug fix changes. 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.

6.4 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 validate 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 6.5. 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 (Figure 6.6) and 0.92 for FreeBSD (Figure 6.7) (with $p\text{-}value < 0.05$). The lower correlation for PostgreSQL can be explained by looking at Figure 6.6. 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

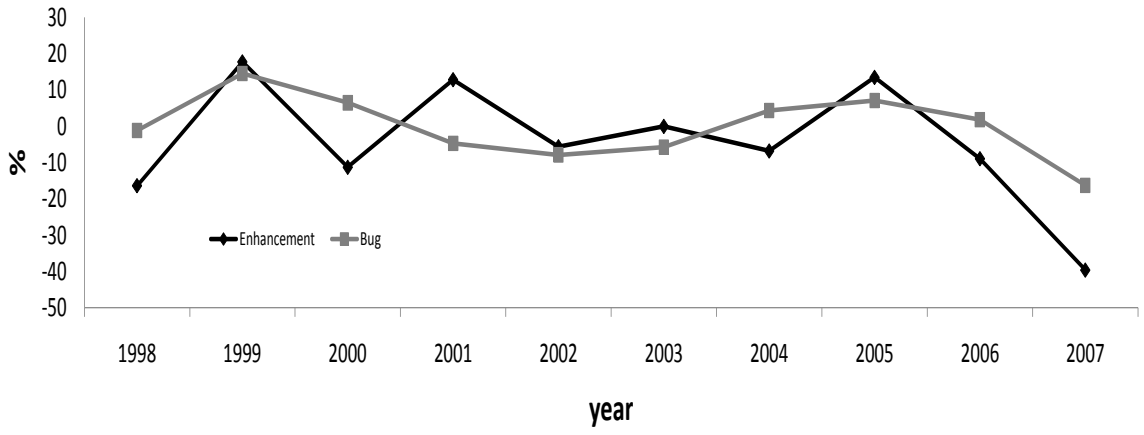


Figure 6.6: Graph showing the correlation between the yearly increase of the **relative** percentage of built-on-new enhancement changes and the yearly increase of the **total** percentage of bug fix changes (the PostgreSQL project).

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 to testing.

Building on recent changes is risky, as more of the development time is spent fixing bugs. Our experiment provides an empirical proof of what seems to be common wisdom in project management.

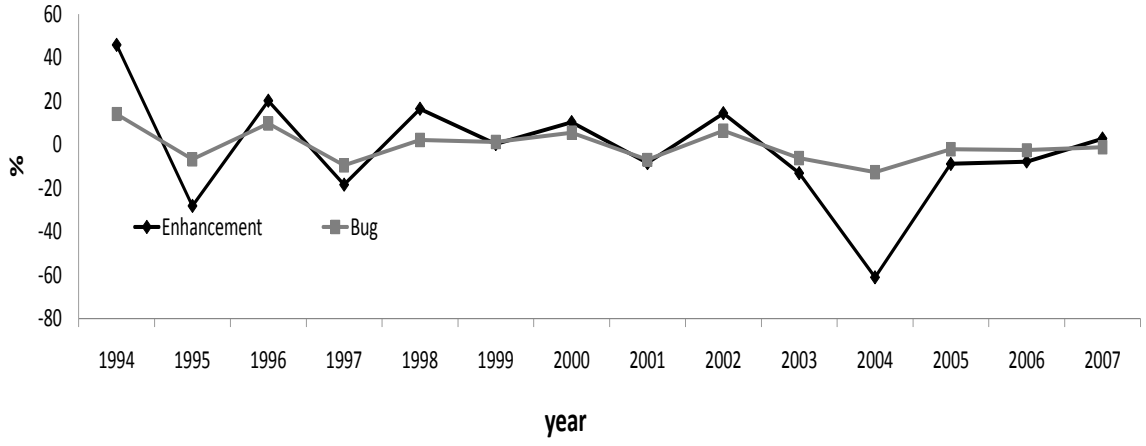


Figure 6.7: Graph showing the correlation between the yearly increase of the **relative** percentage of built-on-new enhancement changes and the yearly increase of the **total** percentage of bug fix changes (the FreeBSD project).

6.5 Chapter conclusion

Tracking the evolution and progress of software projects is usually done by measuring traditional metrics or by informal meetings of the development team. We propose to study the process of developing 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 studied four research questions, two questions are related to project scheduling and the other two questions address principles in project management. In the following, we state the research questions we asked and summary of

the answers:

- **How does the time dependence of *changes* vary over time?**

  Time dependence varies across projects and throughout time. The FreeBSD project has more independent changes than the PostgreSQL project. Both projects build more frequently on newer changes.

- **What is the impact of independent changes?**

  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.

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

  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.

- **Is Building on recent changes risky?**

  Building on recent changes is risky, as more of the development time is spent on fixing bugs. Our experiment provides an empirical proof of what seems to be common wisdom in project management.

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

the next chapter, we discuss how a development period impacts the development of future periods.

Chapter 7

How do periods impact the development in future periods?

In Chapter 6, we studied the time dependence between changes to identify the changes that are build-on-new, built-on-old or independent. In this chapter, we study how much a *period* (e.g. a month or a quarter) as a whole is foundational for the development of future periods. An earlier version of this chapter was published in [3]. In the next chapter, we study the time dependence between subsystems to detect the foundational subsystems.

To study foundational periods, we use the time dependence between periods as described Section 3.2. Identifying the foundational periods would allow managers to turn their attention to the crucial development periods. Managers can put extra effort on testing the code from those periods to ensure that they are risk-free and ensure that the documentation of such periods up-to-date.

In the next sections, we answer three research questions using the concept of time dependence between changes. The first question studies how periods build on older

periods in general. The next question explores if projects build on older periods as they progress. The last question studies the evolution of foundational periods over time.

7.1 How does the time dependence on older *periods* vary over time?

Do projects in general build on old periods, more recent periods or within-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.

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 [10]. 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, as we saw in the previous chapter. 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 each of these backward time dependencies for each quarter. To facilitate our analysis, we graphically rendered this information as a heatmap.

Figure 7.1 is a simple illustration of a heatmap to explain the concept, whereas

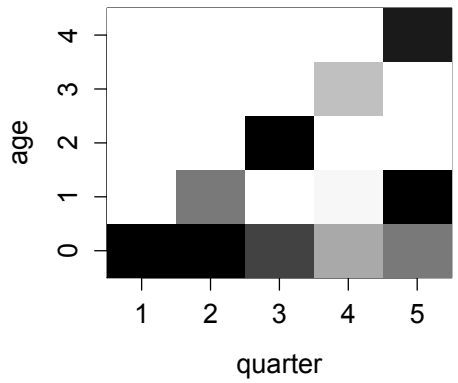


Figure 7.1: Illustration of a heatmap showing the distribution of backward time dependencies over time. Darker cells mean that a period has more backward time dependencies of that age. Age is calculated in quarters.

Figure 7.2 and Figure 7.3 show the actual heatmaps for PostgreSQL and FreeBSD, respectively.

A colored-cell with coordinate (3,2) in Figure 7.1 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, and dominates the heatmaps in Figure 7.2 and Figure 7.3. Therefore, we exclude age zero in Figure 7.4 and Figure 7.5 to make the heatmaps more visible and easy to study (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 7.4 and Figure 7.5 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

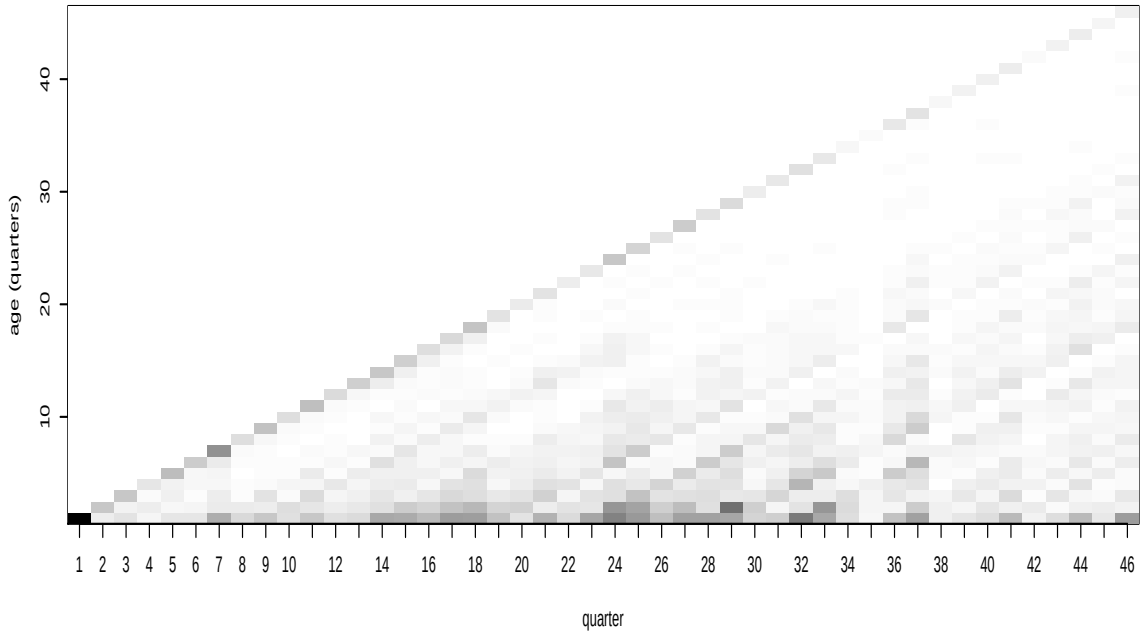


Figure 7.2: Heatmap showing the distribution of backward time dependencies through the lifetime of the PostgreSQL project. For each period (i.e., column), the cells vary in darkness based on the number of backward time dependencies of that age relative to all dependencies for all quarters. Darker cells indicate more dependencies at that age. Age is calculated in quarters. The heatmap contains the within-period time dependence.

distribution of backward time dependencies over time.

Each column of cells in Figure 7.1 shows how the age of the backward time dependencies of a quarter is distributed over time. As shown on Figure 7.4 and Figure 7.5, this distribution is not uniform, and it varies widely across quarters. We briefly summarize our most pertinent findings.

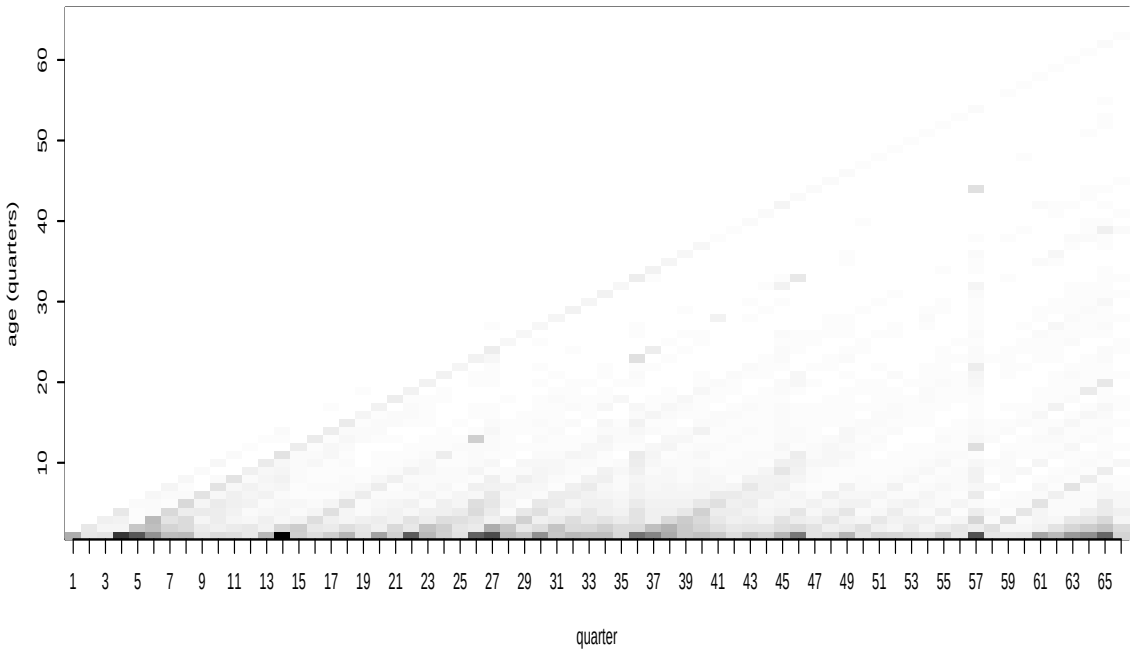


Figure 7.3: Heatmap showing the distribution of backward time dependencies through the lifetime of the FreeBSD project. For each period (i.e., column), the cells vary in darkness based on the number of backward time dependencies of that age relative to all dependencies for all quarters. Darker cells indicate more dependencies at that age. Age is calculated in quarters. The heatmap contains the within-period time dependence.

7.1.1 On average, up to 28% of the backward time dependencies in a quarter are within-period dependencies.

We mentioned that the heatmaps in Figure 7.2 and Figure 7.3 are lightly colored due to the dominance of age zero. We want to study age zero (dependence on the same quarter) further here. Figure 7.6 and Figure 7.7 show that the percentage of within-period backward time dependencies of a period varies between 7% and 100% for PostgreSQL and between 12.8% and 100% for FreeBSD. The averages are 21.2% and 28% respectively. The average age of time dependence gradually decreases for

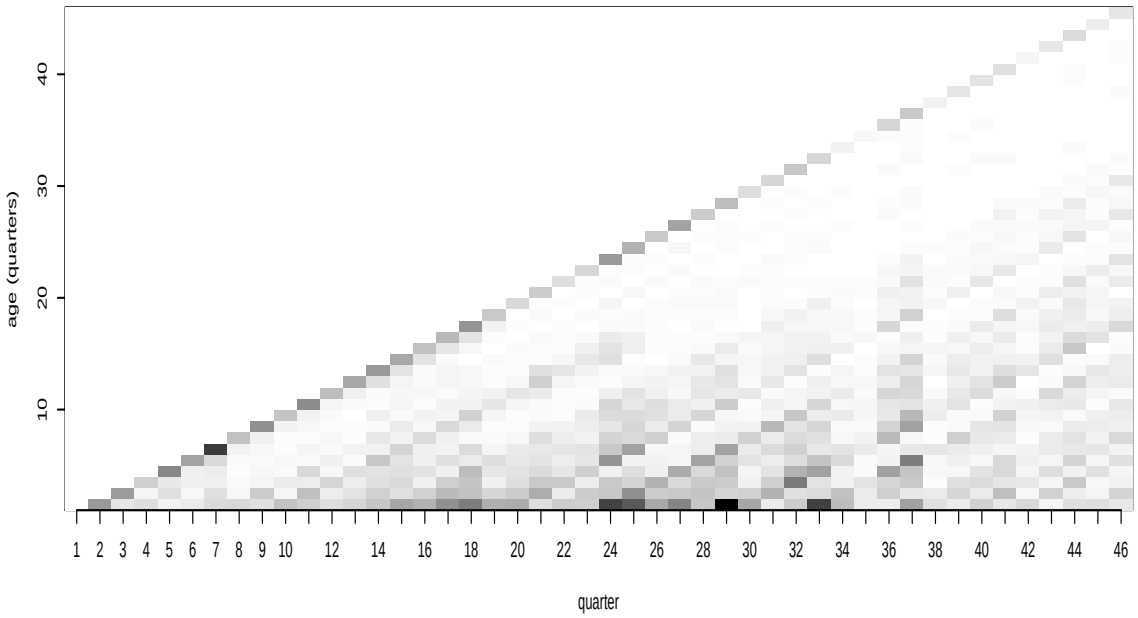


Figure 7.4: Heatmap showing the distribution of backward time dependencies through the lifetime of the PostgreSQL project. For each period (i.e., column), the cells vary in darkness based on the number of backward time dependencies of that age relative to all dependencies for all quarters. Darker cells indicate more dependencies at that age. Age is calculated in quarters. The within-period time dependence is excluded from this heatmap.

quarters after the same quarter as shown in Table 7.1. The second largest average after the same quarter is the immediate last quarter (13.1% for PostgreSQL and 13.28% for FreeBSD). Therefore, the most foundational period for each quarter is the quarter itself.

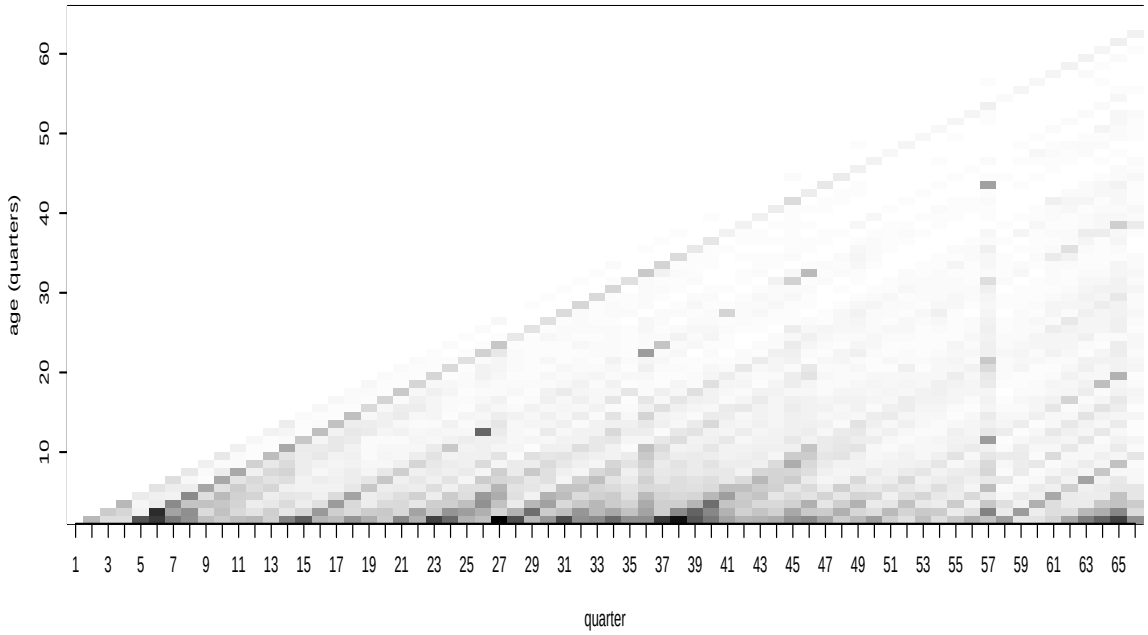


Figure 7.5: Heatmap showing the distribution of backward time dependencies through the lifetime of the FreeBSD project. For each period (i.e., column), the cells vary in darkness based on the number of backward time dependencies of that age relative to all dependencies for all quarters. Darker cells indicate more dependencies at that age. Age is calculated in quarters. The within-period time dependence is excluded from this heatmap.

7.1.2 Quarters with many changes usually build on more changes

Some columns in the heatmaps are very lightly colored compared to the other columns, for example quarter 4 in Figure 7.1, whereas others are very dark. The variance in color is because the corresponding quarters respectively vary in their dependence 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)

quarter	PostgreSQL	FreeBSD
0 (same)	21.2%	27.9%
1	13.1%	13.3%
2	8.5%	8.6%
3	6.6%	6%
4	5.7%	4.9%
5	5%	4%
6	4.5%	3.4%
7	3.7%	2.8%
8	3.2%	2.4%
9	2.9%	2.2%
10	2.7%	1.9%

Table 7.1: Table showing the average time dependence on the same quarter and 10 past quarters for the PostgreSQL and FreeBSD projects.

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 0.60 for PostgreSQL and 0.84 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. PostgreSQL also generally follows the same trend as FreeBSD. However, there are multiple periods in PostgreSQL for which this conjecture does not hold. A large number of changes were introduced in such periods of PostgreSQL, but those changes did not build on older periods.

On average, up to 28% of the changes in a period build on changes that were introduced in the same period. Quarters with many changes generally indicate a higher dependence on older periods.

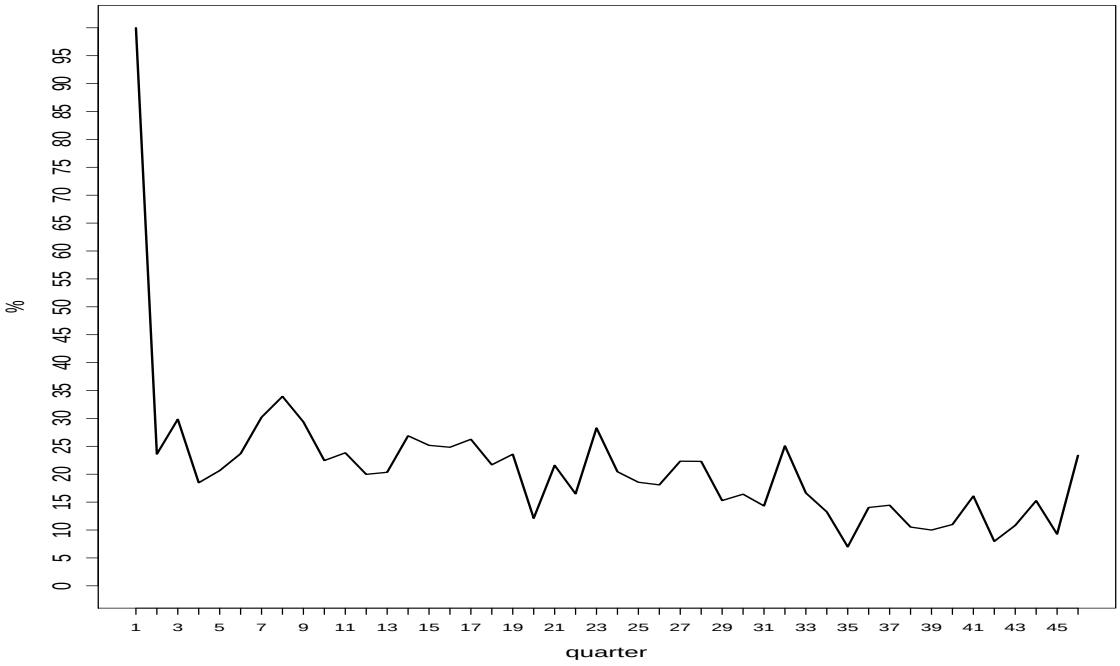


Figure 7.6: The percentage of within-period backward time dependencies of a quarter for the PostgreSQL project.

7.2 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.

Intuitively, we expect that projects become more stable over time. Applied to the topic of this section, 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 will build. To study the evolution of backward

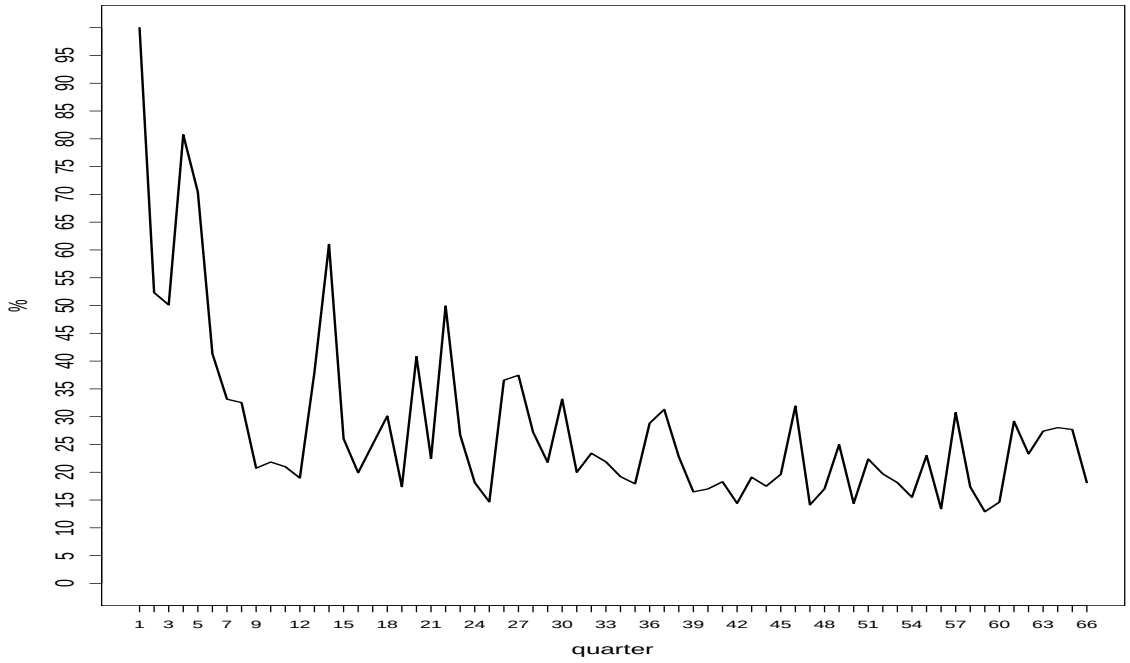


Figure 7.7: The percentage of within-period backward time dependencies of a quarter for the FreeBSD project.

time dependencies over time, we use the boxplots in Figure 7.8 and Figure 7.9, in addition to the heatmaps in Figure 7.4 and Figure 7.5.

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 within-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 quantile. The distance between the lower and upper quantile is called the "Inter-Quartile Range"

(IQR). We briefly explain our findings in the following subsections.

7.2.1 PostgreSQL progressively builds on older periods

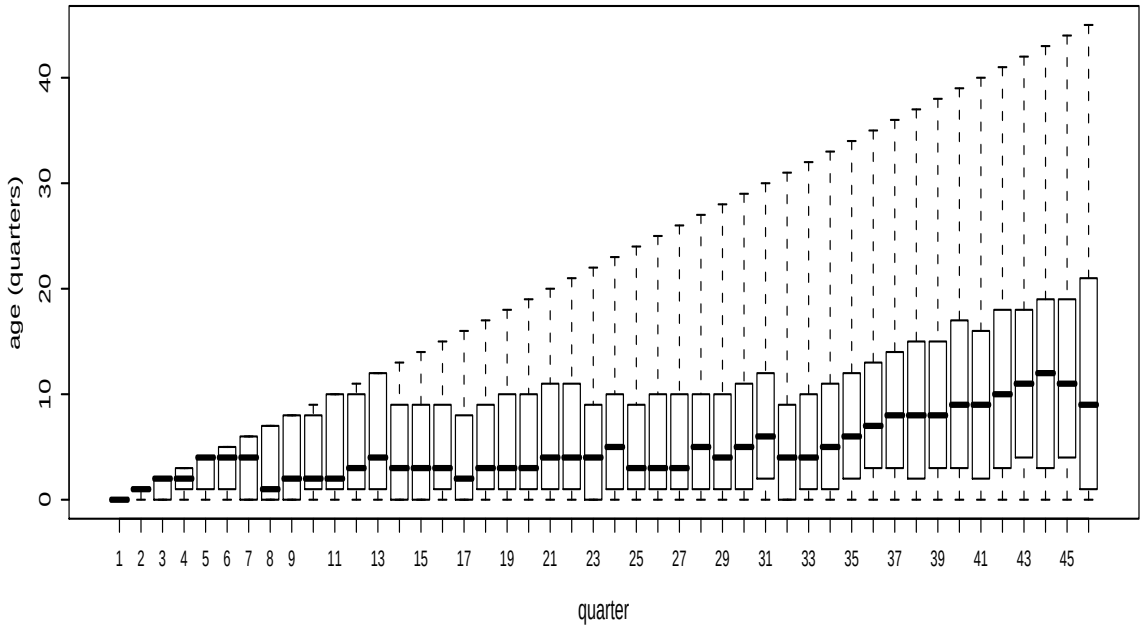


Figure 7.8: Boxplot for the PostgreSQL project showing the minimum, lower quartile, median, upper quartile and maximum of the age of backward time dependencies of a quarter. Age is calculated in quarters.

Backward time dependencies in PostgreSQL are slightly more widespread over time (larger IQR) than those in FreeBSD. In other words, PostgreSQL builds more 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 remains more or less constant, before it starts to fluctuate near quarter 8 and finally starting from quarter 34 it increases steadily (see Figure 7.8). 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.

The increase in the median since quarter 34 means that 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.

7.2.2 FreeBSD periodically cycles between old and recent periods

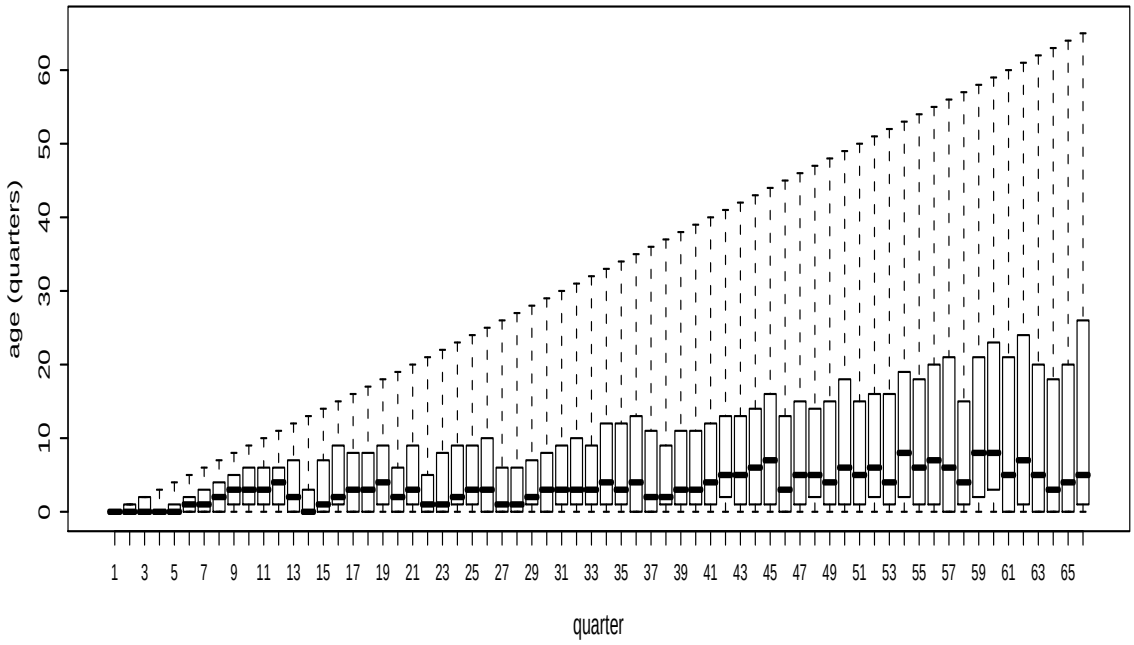


Figure 7.9: Boxplot for the FreeBSD project showing the minimum, lower quartile, median, upper quartile and maximum of the age of backward time dependencies of a quarter. Age is calculated in quarters.

For FreeBSD, the median curve has a very strong periodical character starting from quarter 6 as shown in Figure 7.9. We see periods building on earlier recent

periods (low median) followed by an aging median, which suddenly becomes lower (sometimes 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 decreases again.

Comparing Figure 7.9 with Figure 7.7, we find that the median age of backward time dependencies jumps back to zero (or close to zero) when there are peaks in the relative percentage of within-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.

7.2.3 FreeBSD builds more on recent periods than PostgreSQL

The heatmap of FreeBSD in Figure 7.5 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.

7.2.4 It took 1.5 years before FreeBSD started building on older periods

The periodical aging and renewing of backward time dependencies in FreeBSD only starts from quarter 5. Before quarter 5, FreeBSD almost completely built on within-period changes, as the boxes of boxplots disappear before quarter 5 in Figure 7.9 unlike PostgreSQL. Similarly, darker colors for the FreeBSD heatmap (Figure 7.5) appear later (at quarter 5) than for PostgreSQL (Figure 7.4).

The disappearance of boxes in the FreeBSD boxplot before quarter 5 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 everyone knows 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 [24]. Then there was a lawsuit between Novell and Berkeley after which the 4.3BSD-Lite operating system was deemed contaminated [23]. 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 5, before FreeBSD became stable again.

PostgreSQL progressively builds on older changes, whereas FreeBSD shows a periodical trend in the age of backward time dependencies. It took 1.5 years for FreeBSD to become stable and build on older periods due to a lawsuit between Berkeley and Novell.

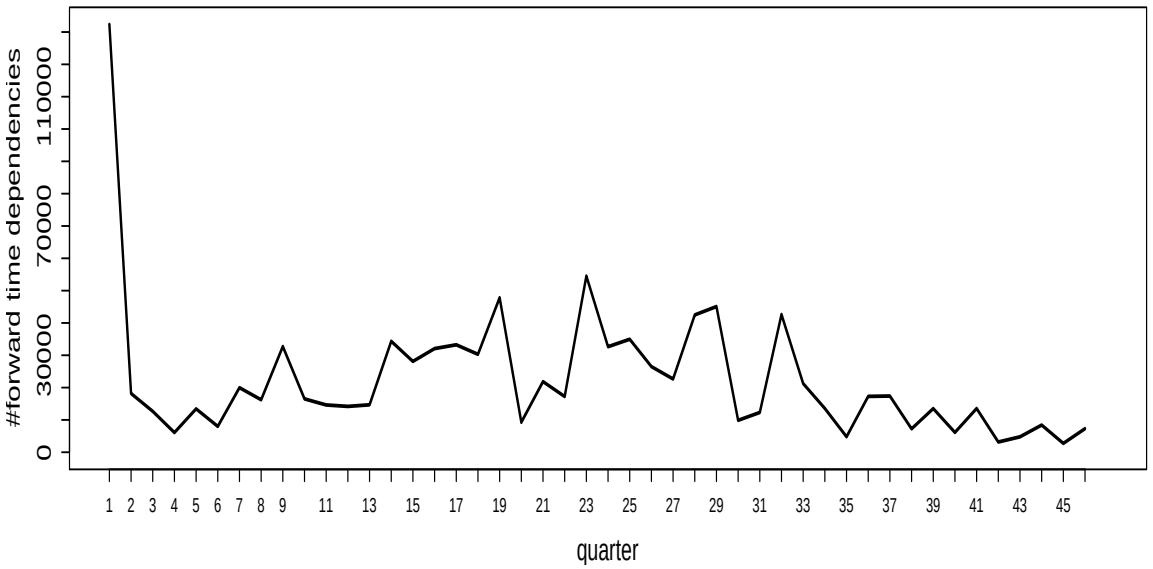


Figure 7.10: The total number of forward time dependencies in quarters for PostgreSQL.

7.3 What are the foundational periods in the 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.

We 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.

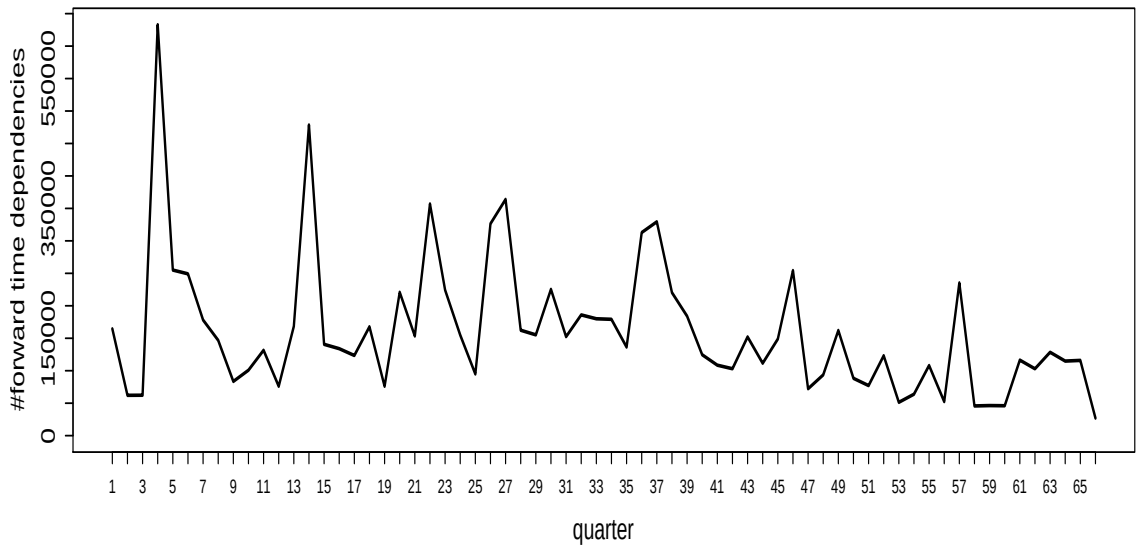


Figure 7.11: The total number of forward time dependencies in quarters for FreeBSD.

To identify the past quarter that a backward time dependence points at (e.g., which quarter the backward time dependencies of age 2 in cell (3,2) point) in Figure 7.1), we need 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 within-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 a diagonal starting in a period is, the more foundational that period is, i.e., later periods keep on building on changes made during that period. Figure 7.4 and Figure 7.5 show some explicit diagonal lines, but also white regions (similar to Figure 7.1). These regions correspond to quarters that are not

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 quarter 24 of PostgreSQL, important parts (e.g., plug-ins and tools) of the source code were extracted from the PostgreSQL code base and were moved to the GBorg (now: PgFoundry) community repository. Globally, FreeBSD has more light periods than PostgreSQL. We discuss our findings below.

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

The heatmaps in Figure 7.4 and Figure 7.5 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 7.10 and Figure 7.11. Foundational quarters show up as peaks in these graphs.

For PostgreSQL, the most foundational periods are quarters 1 and 2, for FreeBSD the first most foundational period 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. The influence of FreeBSD's quarter 5 stems from the 4.3BSD-Lite lawsuit we discussed in Section 6.2. 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. These short diagonals correspond to periods of polishing of less crucial parts of the source code.

7.3.2 Quarters are foundational because of large code imports or invasive changes

Figure 7.10 and Figure 7.11 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 7.5 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 7.10 that PostgreSQL develops less foundational periods starting from quarter 34, when it starts to build on older periods as discussed in section 5.2 and shown in Figure 7.8.

FreeBSD's first foundational quarter (quarter 5) has been discussed earlier. Similar to section 7.2, 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 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.

7.4 Chapter conclusion

New changes often build on older changes, which makes software projects similar to construction projects. Throughout the lifetime of a project, there exist foundational periods during which critical code changes are introduced. Such code changes create the foundational structure on which future code changes and periods build. A good understanding of this hierarchy of temporal dependence can be useful for managers to better plan their projects and to study the evolution of long-lived projects. By knowing such foundational periods, managers can ensure that the archived communication (e.g. mailing lists) is well-kept and that sufficient human expertise is available about these periods.

In this chapter, we studied the foundational periods of two large open source systems and answered the following questions:

- **How does the time dependence on older *periods* vary over time?**

On average, up to 28% of the changes in a period build on changes that were introduced in the same period. Quarters with many changes generally indicate a higher dependence on older periods.

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

  PostgreSQL progressively builds on older changes, whereas FreeBSD shows a periodical trend in the age of backward time dependencies. It took 1.5 years for FreeBSD to become stable and build on older periods due to the lawsuit between Berkeley and Novell.

- **What are the foundational periods in a lifetime of a project?**

  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.

In the next chapter, we proceed with studying how subsystems build on other subsystems.

Chapter 8

How do subsystems build on other subsystems?

So far in this thesis, we studied how changes build on each other and used that to measure the progress of projects. We then proceeded to study time dependence at higher level of abstraction. We studied how a time period in a project's lifetime builds on another period. Time dependence between periods allowed us to study the foundational periods that are crucial in a project's lifetime. This chapter expands the study of time dependence to a fine-grained level. We study the foundational subsystems that appear during different periods.

Knowledge about foundational subsystems would allow managers to delve deeper into the subsystems that are critical for the future development. Managers can concentrate their testing effort on such subsystems. They can make sure that the developers of such subsystems are retained and the documentation of such subsystems is up-to-date. In the next sections, we are going to investigate three research questions about the foundationality of a subsystem.

8.1 How does the influence of foundational subsystems vary over time?

The increase of the size of software repositories makes it harder to identify important parts of a project. Automatically identifying the foundational subsystems would help managers to assign tasks on such subsystems to their experts. Managers will also be able to test these subsystems to make sure they are risk-free, and make sure that the documentation of such subsystems is up-to-date. This section addresses these issues by providing a general overview of the evolution of foundational subsystems. How many foundational subsystems are there? In which time periods do they appear? What are the less foundational subsystems?

To study the evolution of foundational subsystems, we plot spectrographs for PostgreSQL (Figure 8.1) and FreeBSD (Figure 8.2). A spectrograph is a visualization method used in software evolution studies [37, 90, 91]. The horizontal axis of the spectrograph is equally divided into quarters and the vertical axis contains the subsystems sorted by the time of their appearance in the project. The most recently added subsystem appears on top of the spectrograph. Since foundationality is relative, the color compared to other cells is scaled according to the amount of foundationality of a subsystem. The more a quarter becomes dark, the more changes it contains on which other changes build in the future (the more forward time dependence it contains). Figure 8.1 and Figure 8.2 show the spectrographs of PostgreSQL and FreeBSD, respectively. The spectrographs are colored in grayscale to make it easy to print. However, they can be shown in color, which makes it easier to spot the trends. In the following subsections, we study subsystems that are:

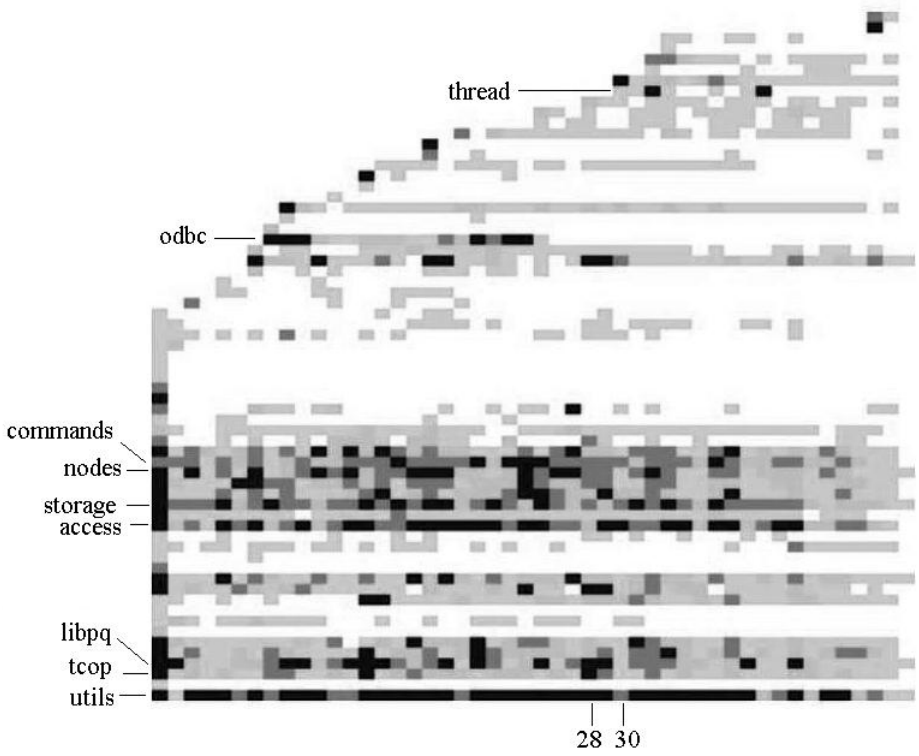


Figure 8.1: Spectrograph of foundational subsystems of PostgreSQL. The horizontal axis is divided into quarters. The vertical axis is divided into subsystems sorted according to the time of their appearance in the system.

- **highly foundational**: subsystems that are very darkly colored in the spectrograph and remain foundational throughout the lifetime.
- **less foundational**: subsystems that are lightly colored and thus contribute less to future development of other subsystems.
- **suddenly foundational**: subsystems that become foundational suddenly at some quarters(s).

8.1.1 Highly foundational subsystems of PostgreSQL

Figure 8.1 shows a spectrograph for the foundationality of changes of PostgreSQL. PostgreSQL consists of 65 subsystems. 36 of the 65 subsystems were introduced in the very first quarter. Many subsystems such as **utils**, **nodes**, **access**, and **storage**, are foundational (dark-colored during the project's lifetime). These subsystems provide core functionalities to PostgreSQL:

- **nodes** provides the structure that allows PostgreSQL to store SQL queries,
- **storage** manages the storage system.

Two subsystems, **utils** and **access**, remain dark-colored most of the time (we study this in detail in the next question). **Utils** provides supporting routines like built-in data types and memory management routines, and contains various utilities to support database transactions and text encoding. The **access** subsystem provides support for queries. These subsystems endured a large number of changes throughout the history of the project and the changes were important for future changes. For example, in quarter 28, 748 commits were done to the **utils** subsystems and 16207 changes throughout PostgreSQL later built on changes from these commits. The memory management utility of the **utils** subsystem is among the subsystems that changed by those commits.

8.1.2 Less foundational subsystems of PostgreSQL

We notice that the color of subsystems becomes lighter as we move to the top of the spectrograph in Figure 8.1 (more recently introduced subsystems). These subsystems are not among the main backend subsystems of PostgreSQL. These subsystems are

small in size like **test** (contains set of test facilities for SQL implementations [71]) and **bin** (where the executable files are stored).

8.1.3 Suddenly foundational subsystems of PostgreSQL

Few subsystems that are located on the top of the spectrograph are darkly colored in a minor number of periods:

- **Thread** (introduced in quarter 30). **Thread** provides a threading API for programmers.

- **Odbc** is an API that enables using PostgreSQL with programming languages on the Windows platform [70]. The **odbc** subsystem became suddenly foundational after the release 3.0 and 3.5 of **odbc** (quarter 8). Those are the major releases for **odbc** that provided various API support and introduced the UNICODE format [26].

These subsystems have important roles in the PostgreSQL project. They provide APIs for various PostgreSQL tasks. However, these subsystems exhibit limited periods of foundationality.

8.1.4 Highly foundational subsystems of FreeBSD

Figure 8.2 shows the spectrograph of 958 subsystems of FreeBSD. The bottom of the spectrograph has a concentration of dark colored subsystems. Two subsystems, **dev** and **kern**, remain darkly colored most of the time. **Kern** is the kernel of FreeBSD operating system. The **dev** subsystem contains the device drivers. The **dev** subsystem changed a lot due to the modifications and additions of drivers. In quarter 28,

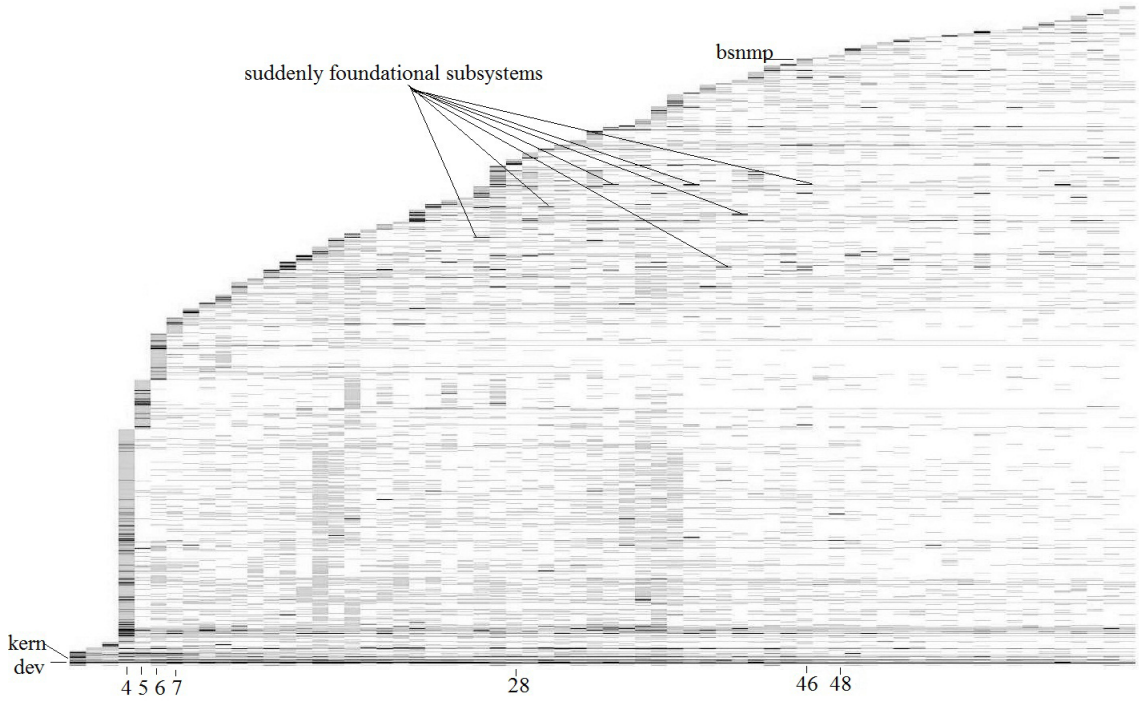


Figure 8.2: Spectrograph of foundational subsystems of FreeBSD. The horizontal axis is divided into quarters. The vertical axis is divided into subsystems sorted according to the time of their appearance in the project.

426 commits have been done for the **dev** subsystem on which 19906 future changes were built. Most of the building changes also belong to the **dev** subsystem itself. The commits to the **dev** subsystem contain changes to drivers like CDR/CDRW, network access and various fixes. There are 12557 future changes that build on the 231 commits to the **kern** in quarter 28. These changes come from subsystems like **vm** (virtual memory), the **fs** (filesystem) and **dev** (drivers). The changes to the kernel include adding new kernel options, enhancing CPU utilization by introducing **SMP** friendly changes and various fixes. One of the kernel functions that was changed is

BUS_ALLOC_RESOURCE(), which is a resource-management function for the device drivers. The **dev** subsystems used this function to change functions in different device drivers. Other subsystems that are concentrated in the bottom dark part are amd64, lib, scsi and i386. A large number of FreeBSD subsystems build on the code in those subsystems. In summary, the highly foundational subsystems (e.g., **kern** and **dev**) are core subsystems in FreeBSD.

8.1.5 Less foundational subsystems of FreeBSD

Figure 8.2 shows a white region comprising a small number of subsystems that are being added in quarters 7 (like **sgsc** and **fib** subsystems). These subsystems after their introduction did not change drastically anymore and the periods in which they were introduced are not darkly colored. In other words, these subsystems are not foundational. We assume that this is because of the transition (inactive) period of FreeBSD around quarter 7, as suggested by the release notes from version 2.0.5 (June 1995). This transition period for FreeBSD was caused by the lawsuit filed by Berkeley. FreeBSD became stable after quarter 7 when its base system was re-written entirely based on the core 4.4 Lite code from UC Berkeley. Subsystems from 4.4 Lite encountered large numbers of changes and many of these subsystems are foundational and darkly colored as described earlier.

8.1.6 Suddenly foundational subsystems of FreeBSD

From quarter 7, FreeBSD develops subsystems that have a small number of foundational periods. The dark colored periods of these subsystems appear scattered across

the lifetime. These subsystems include some separately installable software like textinfo (a documentation system) and ntp (Network Time Protocol), and libraries like libkse and subsystems that contain various kernel source code like arm. Although these subsystems are introduced late, they developed few darkly-colored periods. For example, bsnmp [25] (is a mini-SNMP daemon) has only two highly dark colored periods (46 and 48). These periods were foundational because many changes from the kernel and contributed software build on them. In short, the suddenly foundational subsystems of FreeBSD are mostly system libraries or installable software.

Highly foundational subsystems are core subsystems. The less foundational subsystems are either small in size or are introduced during inactive periods. The suddenly foundational subsystems are usually APIs or systems libraries.

8.2 Are the foundational subsystems stable?

Does a subsystem exhibit limited periods of foundationality, or is it foundational throughout the lifetime of a project? A subsystem that exhibits only limited periods of foundationality is considered very stable, since the subsystems building on it can build on a stable API.

The **stability** of a subsystem is determined by the number of periods that are foundational in that subsystem. We study if a subsystem has few foundational periods or many foundational periods. The fewer periods a subsystem has where its foundationality is concentrated, the more stable it becomes.

We statistically quantify the stability of a subsystem using the basic measures

PostgreSQL		FreeBSD	
Subsystem	Rank	Subsystem	Rank
utils	36	dev	958
nodes	2	kern	957
access	6	i386	16
odbc	1	sys	10
storage	5	gcc	1
libpq	31	user.bin	4
commands	65	libc	13
port	3	netinet	113
optimizer	20	boot	25
parser	10	net	46
catalog	18	gdb	3
executor	19	contrib	19
postmaster	32	binutils	5
tcop	13	pc98	11
pg_dump	16	vm	59
ecpg	11	amd64	99
thread	4	perl5	2
psql	15	libstdc++	7
libpgeasy	7	openssh	22
initdb	14	openssl	12

Table 8.1: Table showing the ranks of stability for the top 20 foundational subsystems. The ranks are out of 65 for PostgreSQL and out of 958 for FreeBSD.

of mean and standard deviation of the foundationality of a subsystem across development periods. By observing the spectrographs in Figure 8.1 and Figure 8.2, we find that stable subsystems have a high variance in foundationality between different development periods. Unstable subsystems on the other hand, have a much smaller variance. Hence, we quantify the stability of a subsystem by subtracting the mean foundationality of a subsystem from the standard deviation.

The larger the standard deviation of a subsystem is compared to its mean (i.e., the larger the value of $STD\text{-}MEAN$), the fewer periods the subsystem has with

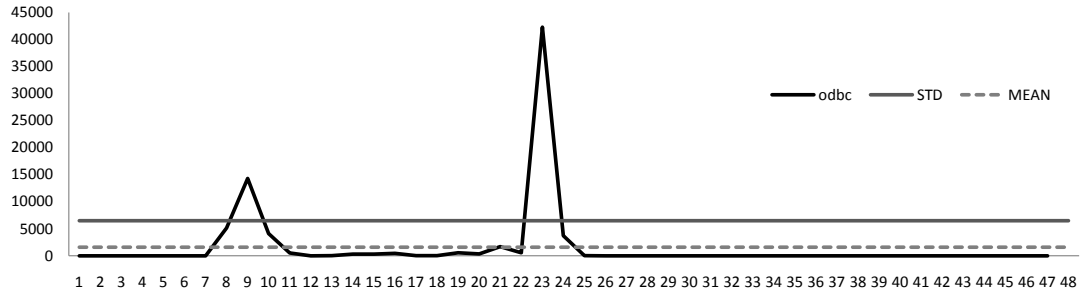


Figure 8.3: The Forward Time Dependence of the **odbc** subsystem in PostgreSQL is more stable as its standard deviation is greater than its mean.

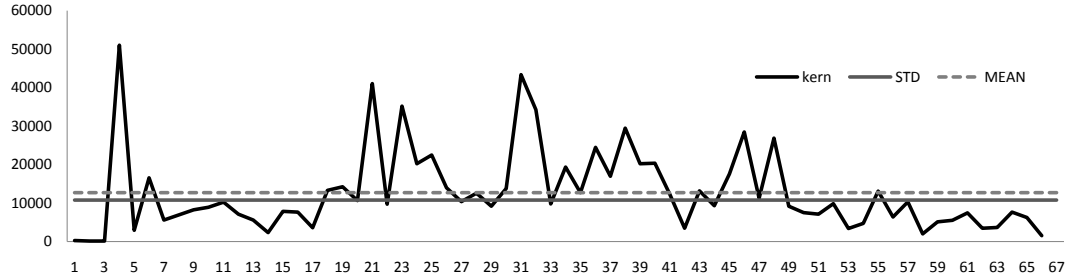


Figure 8.4: The Forward Time Dependence of the **kernel** subsystem in FreeBSD. The **kernel** of FreeBSD is unstable (its standard deviation is less than its mean)

extreme foundationality. Hence, the subsystem becomes more stable, such as the **odbc** subsystem of PostgreSQL (see Figure 8.3). Less stable subsystems have their standard deviation less than, or close to their mean, like the **kernel** of FreeBSD (see Figure 8.4).

Table 8.1 shows the top 20 foundational subsystems and their ranks according to their stability in the project (i.e., based on $STD\text{-}MEAN$). Rank 1 is the most stable. For PostgreSQL, rank 65 is the least stable and rank 958 is the least stable for FreeBSD. The top 20 list of foundational subsystems for both PostgreSQL and FreeBSD contains both the most stable and the least stable subsystems. However,

most of the subsystems in Table 8.1 are either very stable or moderately stable. The least stable subsystem in PostgreSQL is the **commands** subsystem and the most stable one is the **odbc** subsystem. FreeBSD has two subsystems that are ranked very low, the **dev** and the **kern** subsystems.

Table 8.1 shows that the **commands** subsystem of PostgreSQL, and **dev**, and **kern** of FreeBSD are unstable. These are the highly foundational subsystems as discussed in Section 8.1. The rest of the subsystems of FreeBSD (the less foundational and the suddenly foundational ones) have a generally high rank of stability. PostgreSQL has some highly foundational subsystems that are also stable (e.g., **nodes**) and the rank of stability varies from very stable to moderately stable.

We further studied if the relationship between the stability and the foundationality of subsystems holds for all subsystems (not only the top 20). Excluding the three least stable ones (**commands**, **dev**, and **kern**), we find high linear correlation between the foundationality and the stability of subsystems. The Pearson correlation is 0.73 for both PostgreSQL and FreeBSD. Therefore, in general, the more foundational a subsystem, the more stable it is. This means that future subsystems build on changes from fewer periods of the stable subsystem. For example, the **odbc** subsystem of PostgreSQL (which has Rank 1 of stability in Table 8.1) has only two peaks of foundationality in Figure 8.3.

The more foundational a subsystem, the fewer periods it has on which future subsystems mostly build.

8.3 Can we approximate the foundationality of a subsystem using the number of changes to the subsystem?

In Chapter 6, we studied the relation between foundational periods and the number of changes in those periods. Similarly, we expect that if a foundational subsystem undergoes a huge overhaul, other subsystems need to be modified to build on the changed subsystem. However, this does not necessarily hold for non-foundational subsystems. For example, introducing changes to a user interface has no effect on other subsystems. In this question, we study these different possibilities by studying the relation between foundationality of subsystems and the number of changes introduced to these subsystems. To answer this question, we calculate two types of Pearson correlations:

1. We calculate the Pearson correlation between the total number of changes and foundationality for all subsystems across the project lifetime. This tells us if the approximation of foundationality by the total number of changes to a subsystem is a good approximation globally (acorss all subsystems).

2. We calculate for each subsystem the Pearson correlation between the list of the number of changes to a subsystem in each quarter and the list of the foundationality of that subsystem in each quarter. This tells us if the approximation is good for each quarter of individual subsystems.

The global correlation is high: 0.89 for PostgreSQL and 0.88 for FreeBSD. Table 8.2 shows the second correlation for the top 20 foundational subsystems. Overall,

PostgreSQL		FreeBSD	
Subsystem	Correlation	Subsystem	Correlation
utils	0.78	dev	0.71
nodes	1.00	kern	0.62
access	0.78	i386	0.86
odbc	1.00	sys	0.60
storage	0.96	gcc	0.80
libpq	0.96	user.bin	0.99
commands	0.21	libc	0.84
port	0.81	netinet	0.66
optimizer	0.89	boot	0.33
parser	0.92	net	0.40
catalog	0.84	gdb	0.97
executor	0.71	contrib	0.91
postmaster	0.93	binutils	0.97
tcop	0.89	pc98	0.98
pg_dump	0.47	vm	0.80
ecpg	0.72	amd64	0.77
thread	0.90	perl5	0.99
psql	0.80	libstdc++	0.98
libpgeasy	0.88	openssh	0.73
initdb	0.76	openssl	0.84

Table 8.2: Table showing the correlation between the total number of changes and the forward time dependence of top 20 foundational subsystems in each quarter.

the correlation is very high for most of the subsystems, except the following subsystems:

- Subsystems **commands** and **pg_dump** of PostgreSQL. The **commands** subsystem contains the database commands of PostgreSQL and **pg_dump** is a utility to dump the PostgreSQL database into a text file.

- **Boot** and **net** of FreeBSD have low correlations. The **net** subsystem deals with the network interface and boot deals with the booting process.

Although the above subsystems changed a lot, some quarters with fewer changes were more foundational resulting in a low correlation between the total changes and the foundationality. Some of these subsystems are even among the top foundational subsystems in Table 8.1 (e.g., **commands** and **boot**). Although the low correlation is only the case for a minority of the subsystems, using the number of changes instead of foundationality would miss some very important subsystems.

The number of changes introduced to a subsystem is a good approximation for its foundationality. However, some highly foundational subsystems are not detected by measuring the number of changes.

8.4 Chapter conclusion

The growth of software projects in size and complexity makes it hard to analyze these projects entirely. It also prevents the managers to get a clear understanding of the foundational parts of the projects. This chapter proposes a conceptual approach to study the foundationality of subsystems for a given project based on the time dependence between subsystems. Using this approach, we identify the foundational subsystems that create the structure on which future changes build.

Identifying the foundational subsystems could allow managers to ensure that they understand the development activities in those subsystems very well, documentation of such subsystems is up-to-date, and the human expertise of those subsystems is available. By studying the foundational subsystems of the PostgreSQL and FreeBSD projects, we answered the following questions:

- **How does the influence of foundational subsystems vary over time?**

  Highly foundational subsystems are core subsystems. The less foundational subsystems are either small in size or are introduced during inactive periods. The suddenly foundational subsystems are usually APIs or systems libraries.

- **Are the foundational subsystems stable?**

  The more foundational a subsystem, the fewer periods it has on which future subsystems mostly build.

- **Can we approximate the foundationality of a subsystem by the number of changes to the subsystem?**

  The number of changes introduced to a subsystem is a good approximation for its foundationality. However, some highly foundational subsystems are not detected by measuring the number of changes.

Part 3: Conclusion

Chapter 9

Summary and Conclusions

In this thesis, we studied the concept of time dependence which is influenced by the building construction analogy. As in buildings, code changes build on previously-introduced changes. The previously-introduced changes provide the structure on which future changes build. We study the building construction analogy by introducing the concept of time dependence between source code changes. By using time dependence, we studied the progress of projects, how periods impact other periods, and how subsystems build on other subsystems. Using time dependence, practitioners can monitor the evolution of projects, ensure that the communication (e.g., mailing lists) of foundational periods is well-kept and newly-hired people are trained about the foundational subsystems. We applied our approach on over 25 years of software development history. Our findings reiterate that the concept of time dependence is useful in tracking the evolution of software projects. Our work is the first to define time dependence, which is influenced by the building construction analogy.

In this chapter, we conclude our work by specifying its limitations, possible future work and a brief summary of the thesis.

9.1 Limitations and future work

Although we perform our approach on two large systems and validate our findings from the logs and by contacting developers, our approach has some limitations. Here, we specify these limitations, and mention possible ways to address them in the future.

- Our technique to recover the time dependence relations takes into account static dependencies only. Implicit dependencies due to dynamic dependencies are not considered. This may lead to an overestimation of independent changes. In addition to exploring the dynamic dependencies, we would like to explore other types of temporal dependencies, such as those based on cloning and inheritance.

- 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.

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

- 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. We expect that the impact of changes to these dependencies is smaller due to information hiding [68]. We would like to explore this assumption in future work.

- Our case study was done on two open source projects, so our findings may not generalize to commercial projects or other open source 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 historical repository of a project.

- Finally, we define a subsystem as a fixed, top level directory in the projects, similar to approach used by Godfrey et. al. [30] (second level for PostgreSQL and fourth level for FreeBSD as explained in Section 4.2). Although this approach gives us subsystems that are mentioned in the documentation of the projects, we would like to investigate the assumption that subsystems might be scattered across directories at different levels.

9.2 Summary

In this thesis, we propose to study the evolution of a software project as the evolution of a construction project. We validate the hypothesis that we can quantify software evolution using the time dependence between code changes. As in construction projects, software projects consist of changes that build on previously introduced changes. The previously introduced changes provide the structure that the future changes build on. We apply time dependence to study the evolution of software projects along three levels:

- **Change time dependence:** we study if changes are introduced just-in-time

(built on fresh structure), delayed (built on well-established structure), or independent (built on no existing structure). Using our approach, managers can track the progress of their projects similar to how construction managers do. We also study the relation between bug fixing and regular development activities and study the relation between building on new changes and bug appearance.

- **Period time dependence:** Studying how changes build on previous changes allows us to detect foundational periods. Foundational periods are the most contributing periods to the development of future periods. Identifying the foundational periods would allow managers to ensure that the code from those periods is tested to be bug-free, developers of such periods are consulted, and the documentation and the archived communication of these periods are well-kept.

- **Subsystem time dependence:** here we extend our study of foundational periods to a finer level. We study the foundational subsystems that appear at different periods. Managers need to understand the foundational subsystems so they can ensure that those subsystems are tested and the newly-hired people are trained about such subsystems. We study the evolution of foundational subsystems and investigate how stable they are. We also find that the number of changes introduced to a subsystem is a good approximation to its foundationality.

We perform case studies on two large open source projects. We validate our findings from studying the logs from the source code repository and by contacting developers from these projects.

Bibliography

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

[2] Omar Alam, Bram Adams, and Ahmed E. Hassan. Measuring the progress of projects using the time dependence of code changes. In *ICSM '09: Proceedings of the 25th IEEE International Conference on Software Maintenance*, pages 329–338, Edmonton, Canada, 2009. IEEE Computer Society.

[3] Omar Alam, Bram Adams, and Ahmed E. Hassan. A study of the time dependence of code changes. In *WCRE '09: Proceedings of the 16th Working Conference on Reverse Engineering*, pages 21–30, Lille, France, 2009. IEEE Computer Society.

[4] Enrique Alba and J. Francisco Chicano. Software project management with gas. *Information Sciences*, 177(11):2380–2401, 2007.

[5] Christopher Alexander. *The Timeless Way of Building*. Oxford University Press, New York, 1979.

[6] Christopher Alexander, Sara Ishikawa, and Murray Silverstein. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, New York, 1977.

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

[8] Robert Laurence Baber. *Software Reflected: The Socially Responsible Programming of Computers*. Elsevier Science Inc., New York, NY, USA, 1982.

[9] Evelyn J. Barry, Chris F. Kemerer, and Sandra A. Slaughter. On the uniformity of software evolution patterns. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 106–113, Washington, DC, USA, 2003. IEEE Computer Society.

[10] Keith Bennett. Legacy systems: Coping with success. *IEEE Softw.*, 12(1):19–23, 1995.

[11] Abraham Bernstein, Jayalath Ekanayake, and Martin 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.

[12] Salah Bouktif, Giuliano Antoniol, and Ettore 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, pages 155–165, Washington, DC, USA, 2006. IEEE Computer Society.

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

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

[15] Stuart M. Charters, Claire Knight, Nigel Thomas, and Malcolm Munro. Visualisation for informed decision making; from code to components. In *SEKE '02: Proceedings of the 14th international conference on Software engineering and knowledge engineering*, pages 765–772, New York, NY, USA, 2002. ACM.

[16] Annie Chen, Eric Chou, Joshua Wong, Andrew Y. Yao, Qing Zhang, Shao Zhang, and Amir Michail. Cvssearch: Searching through source code using cvs comments. *ICSM '01: Proceedings of the IEEE International Conference on Software Maintenance*, 0:364, 2001.

[17] Lung chun Liu and E. Horowitz. A formal model for software project management. *IEEE Transactions on Software Engineering*, 15:1280–1293, 1989.

[18] Paul Clements, David Garlan, Len Bass, Judith Stafford, Robert Nord, James Ivers, and Reed Little. *Documenting Software Architectures: Views and Beyond*. Pearson Education, 2002.

[19] James O. Coplien. Software design patterns. In *Encyclopedia of Computer Science*, pages 1604–1606. John Wiley and Sons Ltd., Chichester, UK.

[20] Karl Crary. Toward a foundational typed assembly language. In *POPL '03: Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 198–212, New York, NY, USA, 2003. ACM.

[21] Joshua Drake. Personal communication, January 2009. PostgreSQL developer.

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

[23] Freebsd handbook. http://www.freebsd.org/doc/handbook/history.html.

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

[25] Freebsd's bsnmp documentation. http://people.freebsd.org/ harti/bsnmp/.

[26] Odbc versions. http://www.easysoft.com/developer/interfaces/odbc/.

[27] Harald Gall, Mehdi Jazayeri, and Jacek Krajewski. CVS release history data for detecting logical couplings. In *IWPSE '03: Proceedings of the 6th International Workshop on Principles of Software Evolution*, page 13, Washington, DC, USA, 2003. IEEE Computer Society.

[28] Erich Gamma, Richard Helm, Ralph E. Johnson, and John M. Vlissides. Design patterns: Abstraction and reuse of object-oriented design. In *ECOOP '93: Proceedings of the 7th European Conference on Object-Oriented Programming*, pages 406–431, London, UK, 1993. Springer-Verlag.

[29] Daniel M German, Gregorio Robles, and Ahmed E Hassan. Change impact graphs: Determining the impact of prior code changes. volume 0, pages 184–193, Los Alamitos, CA, USA, 2008. IEEE Computer Society.

[30] Michael W. Godfrey and Qiang Tu. Evolution in open source software: A case study. In *ICSM '00: Proceedings of the International Conference on Software Maintenance (ICSM'00)*, page 131, Washington, DC, USA, 2000. IEEE Computer Society.

[31] Todd L. Graves, Alan F. Karr, J. S. Marron, and Harvey Siy. Predicting fault incidence using software change history. *IEEE Transactions on Software Engineering*, 26(7):653–661, 2000.

[32] A.E. Hassan. The road ahead for mining software repositories. In *Frontiers of Software Maintenance, 2008. FoSM 2008.*, pages 48 –57, 28 2008-oct. 4 2008.

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

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

[35] Ahmed E. Hassan and Richard C. Holt. The chaos of software development. In *IWPSE '03: Proceedings of the 6th International Workshop on Principles of Software Evolution*, page 84, Washington, DC, USA, 2003. IEEE Computer Society.

[36] Ahmed E. Hassan and Richard C. Holt. Using development history sticky notes to understand software architecture. *International Conference on Program Comprehension*, 0:183, 2004.

[37] Ahmed E. Hassan, Jingwei Wu, and Richard C. Holt. Visualizing historical data using spectrographs. In *METRICS '05: Proceedings of the 11th IEEE International Software Metrics Symposium*, page 31, Washington, DC, USA, 2005. IEEE Computer Society.

[38] James Herbsleb, David Zubrow, Dennis Goldenson, Will Hayes, and Mark Paulk. Software quality and the capability maturity model. *Commun. ACM*, 40(6):30–40, 1997.

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

[40] Christine Hofmeister, Robert Nord, and Dilip Soni. *Applied software architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[41] Pei Hsia. Making software development visible. *IEEE Software*, 13(3):23–26, 1996.

[42] Watts Humphrey. *Introduction to the personal software process(sm)*. Addison-Wesley Professional, 1996.

[43] Watts S. Humphrey. Characterizing the software process: A maturity framework. *IEEE Software*, 5(2):73–79, 1988.

[44] Philip M. Johnson, Hongbing Kou, Michael Paulding, Qin Zhang, Aaron Kagawa, and Takuya Yamashita. Improving software development management through software project telemetry. *IEEE Software*, 22(4):76–85, 2005.

[45] Magne Jorgensen and Kjetil Molokken-Ostvold. Reasons for software effort estimation error: Impact of respondent role, information collection approach, and data analysis method. *IEEE Transactions on Software Engineering*, 30(12):993–1007, 2004.

[46] Jaak Jurison. Software project management: the manager's view. *Communications of the AIS*, page 2.

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

[48] Claire Knight and Malcolm Munro. Virtual but visible software. In *IV '00: Proceedings of the International Conference on Information Visualisation*, page 198, Washington, DC, USA, 2000. IEEE Computer Society.

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

[50] Jay Kothari, Ali Shokoufandeh, Spiros Mancoridis, and Ahmed E. Hassan. Studying the evolution of software systems using change clusters. In *ICPC '06: Proceedings of the 14th IEEE International Conference on Program Comprehension*, pages 46–55, Washington, DC, USA, 2006. IEEE Computer Society.

[51] Robert E. Kraut and Lynn A. Streeter. Coordination in software development. *Communications of the ACM*, 38(3):69–81, 1995.

[52] Philippe Kruchten. The 4+1 view model of architecture. *IEEE Softw.*, 12(6):42–50, 1995.

[53] Phipippe Kruchten, Bran Selic, Grant Larsen, and Alan Brown. Describing software architecture with uml. In *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, page 777, Washington, DC, USA, 2001. IEEE Computer Society.

[54] Nathan LaBelle and Eugene Wallingford. Inter-package dependency networks in open-source software. *CoRR*, cs.SE/0411096, 2004.

[55] 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*, page 20, Washington, DC, USA, 1997. IEEE Computer Society.

[56] Marek Leszak, Dewayne E. Perry, and Dieter Stoll. Classification and evaluation of defects in a project retrospective. *Journal of Systems and Software*, 61(3):173–187, 2002.

[57] Harvey Maylor. Beyond the gantt chart: Project management moving on. *European Management Journal*, 19(1):92 – 100, 2001.

[58] Thilo Mende, Rainer Koschke, and Felix Beckwermert. An evaluation of code similarity identification for the grow-and-prune model. *J. Softw. Maint. Evol.*, 21(2):143–169, 2009.

[59] T. Mens, J. Fernandez-Ramil, and S. Degrandsart. The evolution of eclipse. In *IEEE International Conference on Software Maintenance, 2008. ICSM 2008*, pages 386–395. IEEE Computer Society, 28 2008-Oct. 4 2008.

[60] E. Merlo, M. Dagenais, P. Bachand, J.S. Sormani, S. Gradara, and G. Antoniol. Investigating large software system evolution: the linux kernel. In *Computer Software and Applications Conference, 2002. COMPSAC 2002. Proceedings. 26th Annual International*, pages 421 – 426, 2002.

[61] Siavash Mirarab, Alaa Hassouna, and Ladan 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*, pages 177–188, Washington, DC, USA, 2007. IEEE Computer Society.

[62] Audris Mockus and Lawrence G. Votta. Identifying reasons for software changes using historic databases. In *ICSM '00: Proceedings of the International Conference on Software Maintenance (ICSM'00)*, page 120, Washington, DC, USA, 2000. IEEE Computer Society.

[63] Audris Mockus, David M. Weiss, and Ping Zhang. Understanding and predicting effort in software projects. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 274–284, Washington, DC, USA, 2003. IEEE Computer Society.

[64] Christopher R. Myers. Software systems as complex networks: Structure, function, and evolvability of software collaboration graphs. *Physical Review E*, 68(4):046116, Oct 2003.

[65] Nachiappan Nagappan and Thomas Ball. Use of relative code churn measures to predict system defect density. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 284–292, New York, NY, USA, 2005. ACM.

[66] Thomas Panas, Rebecca Berrigan, and John Grundy. A 3d metaphor for software production visualization. *International Conference on Information Visualisation, 2003*, 0:314.

[67] Thomas Panas, Rüdiger Lincke, and Welf Löwe. Online-configuration of software visualizations with vizz3d. In *SoftVis '05: Proceedings of the 2005 ACM symposium on Software visualization*, pages 173–182, New York, NY, USA, 2005. ACM.

[68] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[69] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *SIGSOFT Software Engineering Notes*, 17(4):40–52, 1992.

[70] PostgreSQL developer handbook. http://www.postgresql.org/developer/ext.backend_dirs.html

[71] PostgreSQL release notes. http://www.postgresql.org/docs/8.2/static/release.html.

[72] WWISA philosophy. 1999, worldwide institute of software architects. http://www.wwisa.org/.

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

[74] Wolfgang Pree. *Design patterns for object-oriented software development*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1995.

[75] John J. Rakos. *Software project management for small to medium sized projects*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990.

[76] David Robertson and Jaumi Agustí. *Software blueprints: lightweight uses of logic in conceptual modeling*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1999.

[77] Gregorio Robles, Juan Jose Amor, Jesus M. Gonzalez-Barahona, and Israel Herraiz. Evolution and growth in large libre software projects. In *IWPSE '05: Proceedings of the Eighth International Workshop on Principles of Software Evolution*, pages 165–174, Washington, DC, USA, 2005. IEEE Computer Society.

[78] Douglas C. Schmidt. Using design patterns to develop reusable object-oriented communication software. *Communications of the ACM*, 38(10):65–74, 1995.

[79] Marc T. Sewell and Laura Sewell. *The Software Architect's Profession: An Introduction*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

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

[81] S. Valverde and R. V. Sole. Hierarchical Small Worlds in Software Architecture. *ArXiv Condensed Matter e-prints*, 2003.

[82] Michiel van Genuchten. Why is software late? an empirical study of reasons for delay in software development. *IEEE Transactions on Software Engineering*, 17(6):582–590, 1991.

[83] Georg von Krogh, Sebastian Spaeth, and Karim R. Lakhani. Community, joining, and specialization in open source software innovation: a case study. *Research Policy*, 32(7):1217 – 1241, 2003. Open Source Software Development.

[84] Yi Wang, Defeng Guo, and Huihui Shi. Measuring the evolution of open source software systems with their communities. *SIGSOFT Software Engineering Notes*, 32(6):7, 2007.

[85] Lian Wen, Diana Kirk, and R. G. Dromey. Software systems as complex networks. In *COGINF '07: Proceedings of the 6th IEEE International Conference on Cognitive Informatics*, pages 106–115, Washington, DC, USA, 2007. IEEE Computer Society.

[86] Michel Wermelinger and Yijun Yu. Analyzing the evolution of eclipse plugins. In *MSR '08: Proceedings of the 2008 international working conference on Mining software repositories*, pages 133–136, New York, NY, USA, 2008. ACM.

[87] Richard Wettel. Visual exploration of large-scale evolving software. In *ICSE Companion*, pages 391–394, 2009.

[88] Richard Wettel and Michele Lanza. Visual exploration of large-scale system evolution. In *WCRE '08: Proceedings of the 2008 15th Working Conference on Reverse Engineering*, pages 219–228, Washington, DC, USA, 2008. IEEE Computer Society.

[89] Richard Wettel and Michele Lanza. Visually localizing design problems with disharmony maps. In *SoftVis '08: Proceedings of the 4th ACM symposium on Software visualization*, pages 155–164, New York, NY, USA, 2008. ACM.

[90] Jingwei Wu, Richard C. Holt, and Ahmed E. Hassan. Exploring software evolution using spectrographs. In *WCRE '04: Proceedings of the 11th Working Conference on Reverse Engineering*, pages 80–89, Washington, DC, USA, 2004. IEEE Computer Society.

[91] Jingwei Wu, Claus W. Spitzer, Ahmed E. Hassan, and Richard C. Holt. Evolution spectrographs: Visualizing punctuated change in software evolution. In *IWPSE '04: Proceedings of the Principles of Software Evolution, 7th International Workshop*, pages 57–66, Washington, DC, USA, 2004. IEEE Computer Society.

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

[93] T.-J. Yu, V. Y. Shen, and H. E. Dunsmore. An analysis of several software defect models. *IEEE Transactions on Software Engineering*, 14(9):1261–1270, 1988.

[94] John A. Zachman. A framework for information systems architecture. *IBM Systems Journal*, 26(3):276–292, 1987.

[95] Yu Zhou, Michael Würsch, Emanuel Giger, Harald C. Gall, and Jian Lü. A bayesian network based approach for change coupling prediction. In *WCRE '08: Proceedings of the 2008 15th Working Conference on Reverse Engineering*, pages 27–36, Washington, DC, USA, 2008. IEEE Computer Society.

[96] Thomas Zimmermann, Peter Weisgerber, Stephan Diehl, and Andreas Zeller. Mining version histories to guide software changes. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 563–572, Washington, DC, USA, 2004. IEEE Computer Society.