THE IMPACT OF KNOWLEDGE LOSS ON SOFTWARE PROJECTS: TURNOVER, CUSTOMER FOUND DEFECTS, AND DORMANT FILES

Samuel M. Donadelli

A thesis
in
The Department
of
Computer Science

Presented in Partial Fulfillment of the Requirements
For the Degree of Master in Applied Sciences Software Engineering
Concordia University
Montréal, Québec, Canada

April 2015

Concordia University
School of Graduate Studies

This is to certify that the thesis prepared

By: **Samuel M. Donadelli**

Entitled: **The impact of knowledge loss on software projects: turnover, customer found defects, and dormant files**

and submitted in partial fulfillment of the requirements for the degree of

Master in Applied Sciences Software Engineering

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining commitee:

Dr. D. Goswami _____ Chair

Dr. J. Rilling _____ Examiner

Dr. E. Shihab _____ Examiner

_____ Examiner

Dr. P. Rigby _____ Supervisor

Approved _____
Chair of Department or Graduate Program Director

_____ 20 _____ _____
Dean
Faculty of Engineering and Computer Science

Abstract

The impact of knowledge loss on software projects: turnover, customer found defects, and dormant files

Samuel M. Donadelli

The success of a software project is dependent on the expertise and knowledge of its developers. In this dissertation, we use empirical studies to develop an understanding of the impact of knowledge loss on software projects. First, we studied the damage done to projects from turnover, the susceptibility of the project to future turnover, and the suggestion of potential successors to assume abandoned files. Based on the project vulnerability to turnover, project leaders can induce key developers to stay with the project and to mitigate files abandonment. Second, we did an empirical research on the impact of turnover on the quality of a software project. Third, we performed an examination of the impact of inactive files (dormant files).

Our findings on the first research topic showed that the greater the spread of knowledge the less likely a project is to be affected by turnover. Moreover, we found that knowledgeable developers, rather than newcomers, take over abandoned code. In our second study, we observed an unexpected result that in the Chrome web-browser project, the number of developers who leave and join both decreased the number of post-release defects. We discuss this unexpected result. The third study on dormant files, i.e. inactive files, contrasted a legacy system with a popular system. We found that for a legacy system, the developers that take on dormant files were experienced developers.

Acknowledgments

At the conclusion of a stage, you must thank those who have been by our side, because, without support, we are helpless.

In particular to God who gives me hope and faith, in every moment of my life.

Professor Peter Rigby, my thesis supervisor, who have taught me research methodology and supported me in my research.

Thanks to all my lab mates Rupak, Shams, Louis, Murtuza and Latifa for all the knowledge and experiences we shared together. Special thanks to Yuecai for all the help and support in a statistical point of view.

Thanks to the people from other labs Davoud, Everton, Giri, Andy and others, for the times they accepted me in their labs. I was always there to share a good conversation, in my break times, and we talked about everyday life, courses, programming. Special thanks to Professor Tsantalis for such nice conversations we had during my research period.

To Concordia University, for the opportunity to perform my Master in Applied Sciences Software Engineering and to the Canadian government for provided me through my advisor's grants.

My family, even being so far from me, we are always together in my heart and thoughts. My special thanks to Paulo, my love, who gave me strength and wise advice each and every day of my life. Finally, thanks to my friends Zara, Abel, Francy and Cyrille, that were always there to listen and share their thoughts.

Contribution of Authors

I had the pleasure to work with Yue Cai Zhu during my Master Thesis research. He assisted me in the following parts:

- Chapter 2 - Developer turnover and succession Empirical studies of the susceptibility and damage from developer turnover. He developed the truck factor algorithm and wrote the Section 2.4.
- Chapter 3 - Organizational volatility and post-release defects: A replication case study using data from Google Chrome. He assisted with the statistical model selection (Section 3.2.2).
- Chapter 4 - A preliminary examination of dormant files on software projects. He assisted with the idea of the figure "Dormant file lifecycle".

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Know-how is vital to any organization. The loss of knowledge can reduce a company's efficiency and effectiveness. The quality of software projects is affected by the reduction of developers in the team. To gauge the employee retention in a company there is the turnover rate which measures the joining and leaving of employees in an organization. The know-how is also affected when a particular task is performed infrequently, the knowledge rate decreases considerably tending to zero [31]. The factors mentioned above influence the loss of know-how in a software project, in other words, the software projects lose their competitive edge.

Knowledge loss happens when a software developer leaves the project. This know-how gap is manifested as abandoned files on the software system. Because the files are abandoned, they are harder to be maintained. In the first chapter, we study three aspects of knowledge loss. (1) damage, (2) risk of future turnover, and (3) potential successors. Based on the turnover rate and the number of abandoned files,

it is important to discover what is the damage done to projects from turnover, to obtain a better understanding of the susceptibility of project to future turnover, and to discover potential successors who can take over the abandoned files. The greater the turnover rate the greater the risk of knowledge loss.

In Chapter 2, we focus on turnover and software quality. The software quality, as measured by the number of defects that customers experience, is affected by the knowledge loss created by turnover and other organizational changes. In previous research, Mockus *et al.* [19] found that the greater the number of leaving developers in a software project, the greater was the number of customer found defects. Considering that Mockus research was performed in a volatile environment in which layoffs were frequent, it is important to understand the effect of turnover in other environments. For instance, in a context of a growing and successful OSS such as Google Chrome. It appears that, the greater the number of leaving developers in Chrome, the lower the number of customers found defects. We discuss possible explanations for this result. The impact on the quality of a software project is dependent on the context in which the project is included.

Since knowledge loss also occurs when a task is not worked regularly, in Chapter 3 we look at how familiar a developer is with source files. The research about forgetting curve suggests that if students do not review information regularly they will forget the information as time goes by [31]. Adapting this context to software development, the code that does not change regularly will be forgotten. Long time inactive files (dormant files) will be hardly remembered because developers no longer understand

the file and its relationships to other files in the system. The goals of studying dormant files are to observe their amount on the project, the presence of developers who worked on dormant files, the experience of people who take over these files, and dormant files bugs. We find that the longer the file is inactive the greater the decreasing of knowledge rate.

1.1 Research Statement and Overall Methodology

In Figure 1.1, we observe the four objectives in this thesis. We used five distinct Open Source Software projects (OSS) to comprehend turnover, organizational volatility, and dormant files. We performed each of this studies using different data sets (i.e. review project documents, project history) and different approaches (i.e. statistical and grounded theory analyzes). By the use of this diverse material and techniques, we are able to enhance the generality and reliability of our contribution [37].

1.2 Outline of the thesis

The chapters of this thesis have distinct sections for literature, research questions, methodology and data, and results. We believe that this structure is suitable for the purpose of each chapter (see Figure 1.1).

Developer Turnover and Succession: Empirical studies of the susceptibility and damage from developer turnover - Chapter 2

Motivation: The success of a software project is dependent on the expertise and

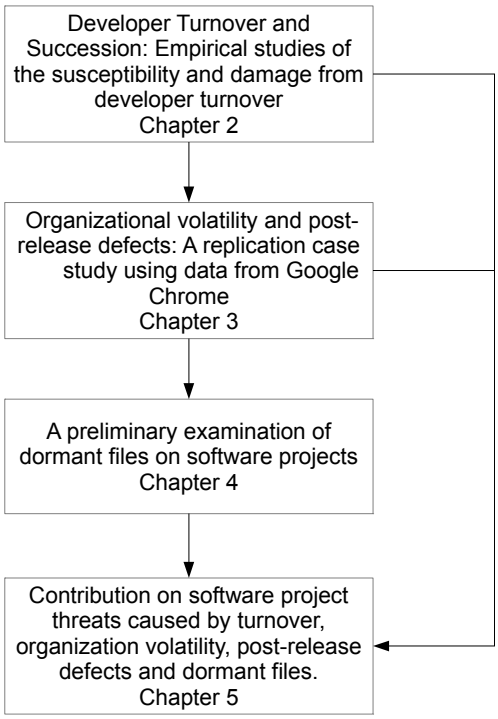

Figure 1.1: Stages in the Research Process

knowledge of its developers. When a developer leaves a project the team losses that developer's knowledge. The remaining developers must take over the maintenance of the code the leaver abandoned.

Research questions: (Q1) What is the rate of developer turnover? How much knowledge loss has occurred as a result of turnover? (Q2) How susceptible is a project to future turnover and knowledge Loss? (Q3) Can we mitigate the impact of developer turnover by suggesting potential successors?

Literature: We reviewed the literature on code abandonment, turnover in software projects, succession, truck factor and coordination requirements.

Metodology: First, we observed the turnover rate on software projects divided by periods of three months. Second, we used an enhanced truck factor algorithm to calculate the project damage caused by turnover. Finally, we used an expertise matrix calculation to define potential successors to abandoned code.

Outcome: We find large successful project can function with an abandonment ratio of up to 22%. We provide an algorithm that can calculate the hypothetical maximum knowledge loss when a group of developers leaves the projects. We introduce a stopping condition and prove that our algorithm finds the optimal solution. Given a group size of 10 and a team size of 250 developers we calculate the knowledge loss with 13 orders of magnitude fewer calculations than previous work – our algorithm works on large projects. We find that instead of newcomers, experienced developers who have been with the project for multiple years take over the maintenance of abandoned files. As a result, we are able to accurately suggest successors based on the

files that have co-changed with the abandoned file.

Organizational volatility and post- release defects: A replication case study using data from Google Chrome - Chapter 3

Motivation: The quality of software projects is affected by developer turnover. Mockus [19] studied organizational volatility in the context a large switching software project at Avaya. We replicate his model of the impact of organizational volatility on post-release defects. At the time of Mockus's study, Avaya was experimenting with outsourcing and layoffs were prevalent. In contrast, we study volatility on the Chrome web-browser, which is growing rapidly in terms of popularity and team size.

Research questions: (Q1) Is the number of changes that are made to a code related to the number of defects found in it? (Q2) Does the number of developers that touch a software artifact influence the number of defects found in it? (Q3) When a project loses a developer, this project loses knowledge. Does it affect the number of found defects? (Q4) Do newcomers bring fresh ideas to the project? (Q5) Does the number of co-changing files increases the number of found defects in a file? (Q6) Does the experience and the expertize of a developer impact on the number of defects found in a file?

Literature: We reviewed literature on abandonment of code, turnover in software projects, prediction models of defects, software dependencies, the impact of churn on software defects.

Metodology: The history of the software was divided per releases. For each

release, we looked the past development branch to calculate the dependent variables, and we looked for the independent variable (the customer found defects) from the date of the release until the start date of the next release. Finally, we were able to build our prediction model.

Outcome: The greater the complexity, the number of developers working on the file, the greater the number of defects. The greater the number of newcomers, leavers and developer experience, the lower the number of defects. The former findings agree with the literature, while the latter are suprising. We suggest two possible explanations that deserve future work. First, when a developer leaves the project, the features they were working on may be put on hold, which would lead to fewer changes and fewer defects. Second, existing co-owners may take over the leaving developers work leading to a more focused set of changes and fewer defects.

A preliminary examination of dormant files on software projects - Chapter 4

Motivation: Files that are inactive for a long period (dormant files) will be hardly remembered by the developers who worked with these files.

Research questions: (Q1) How many files are dormant and how many LOCs are contained in dormant files? (Q2) When will a file become dormant? How many dormant files become active and how many of them remain dormant? How long does it take for a dormant file to become active again? (Q3) How much experience does a developer who works on a dormant file have? (Q4) How many dormant files have surprise bugs?

Literature: We reviewed the literature on legacy systems, prediction of defects based on software history, psychology regarding forgetting curve, and surprise defects.

Metodology: We identified the dormant files as per their difference in days between each commit. We obtained the LOC using a script that counts LOC per file. The experience of a developer was considered, from the first date the developer participated in the project to the date of the commit. The bug extraction was done by mining the bug database and linking each bug number with the respective commit.

Outcome: The legacy system used in this research, Evolution OSS, is more prone to dormant files. Moreover, the complexity of dormant files in the legacy system accounted for 80% of all complexity. The positive side is despite the fact that the systems we studied had dormant files, the developers that assumed these dormant files were experienced to the project.

Discussion and Conclusion - Chapter 5

The main objective of this work is the contribution on threats of knowledge management on software project caused by turnover, customer found defects, and dormant files. This contribution can be described as:

The greater the spread of knowledge the less likely a project is to be effect by turnover. Knowledgable developers, rather than newcomers, take over abandondned code. We observed an unexpected result that in Chrome web-browser project, the number of developers who leave and join both reduce the number of post-release defects. Our findings on the dormant files study, showed that for the legacy system, the proportion of files in the system is that the dormant files was greater than the

active files. Futhermore, the developers who assume dormant files were experienced developers.

Chapter 2

Developer Turnover and Succession: Empirical studies of the susceptibility and damage from developer turnover

In the previous chapter, we presented our research goals. In this chapter, we verify the following aspects of turnover in software projects: the knowledge loss caused by turnover, the susceptibility of project to turnover and we observe the possibility to solve the lack of knowledge caused by the leaving developer.

High turnover rates have been shown to lead to a decrease in a firm's productivity [14]. Like other jobs that involve knowledge workers, when a software developer leaves there is a knowledge gap. This gap is manifested as abandoned files on the software system that can be difficult to maintain. Turnover has also been shown to decrease the quality of software products [17]. As a predictor of future defects the

number of developers to abandon a file is second only to the number of changes to a file [23].

Turnover reduces the spread of knowledge. In the worst case, when a single developer controls the entire system or the most critical parts of a large system their loss is catastrophic. In the Agile community the spread of knowledge across the development team is colloquially know as the 'truck factor' – the number of developers that must leave (*e.g.,* get hit by a truck) before the project becomes unsustainable [38, 34]. Agile development practices, such as pair programming, have the consequence of ensuring that no single developer hold the knowledge of a file exclusively [36]. Similarly, peer review practices expose developers to parts of the system that they would otherwise not have seen thereby reducing the potential for knowledge loss[28, 27].

The goal of this chapter is to first quantify the damage that has been done by past turnover, to quantify the knowledge spread and exposure to future turnover, and to mitigate the impact of turnover by suggesting successors for abandoned code.

2.1 Research Questions

We answer the following research questions to understand the impact of knowledge loss on software projects.

RQ1, Damage from Past Turnover: What is the rate of developer turnover? How much knowledge loss has occurred as a result of turnover?

Maintaining abandoned code is difficult because the team lacks knowledge of its creation and structure. We gauge the rate of turnover on projects and measure the proportion of files that have been abandoned. We contrast the turnover ratio among projects and compare it with other industries. These basic results frame our subsequent findings.

RQ2, Truck Factor: How susceptible is a project to future turnover and knowledge loss?

Previous work has calculated the maximum number of files that are lost when a group of developers leaves. Since this calculation has a time complexity of $\binom{n}{g}$ for all cases, they were able to examine small projects only. We contribute with a stopping condition that allows us to calculate the maximum loss quickly, so that we can examine the truck factor on large projects. We prove that we find the optimal solution. We measure the potential for knowledge loss on Linux and Chrome.

RQ3, Successors: Can we mitigate the impact of developer turnover by suggesting potential successors?

In the previous research questions, we addressed the impact and future risk of turnover, but what should a project do when a developer leaves? There have been many excellent studies of introducing new developers to a project [1, 40, 19, 5]. However, we find that when a file is abandoned the developer who takes it over has multiple years of experience. As a result, we use a modified version of Cataldo *et al.*'s [7] coordination requirements matrix to suggest developers who have worked in similar areas as potential successors instead of new developers. We measure how many

possible successors exist for each abandoned file on the system and evaluate how well our technique predicts actual successors of abandoned files.

The remainder of this chapter is structured as follows. In Section 2.2, we describe our methodology and dataset. In Section 2.3, we measure the impact of past turnover. This section is the background necessary for the subsequent sections. In Section 2.4 we prove that our truck factor algorithm finds the optimal solution and present the maximum knowledge loss that occurs when a group of developers leaves. In Section 2.5, we show that most abandoned files are adopted by expert developers and suggest possible successors to abandoned files. In the final two sections, we discuss threats to validity, future work and conclude the chapter.

2.2 Methodology, and Data

Knowledge loss occurs when a developer leaves a project. Since we do not have the official records of when a developer joins and leaves a project, we consider a developer to have left the project when they make their last commit. We exclude the final year of development to avoid mistakenly assuming a developer has left the project when they have simply been inactive [16].

To resolve duplicate email addresses to a single individual, we use Canfora *et al.* [5] name aliasing tool. We added an additional cleaning stage where diacritics are converted into their ASCII form as their tool cannot handle these characters (*e.g.,* ŏ is converted to o).

File ownership: The amount of knowledge lost when a developer leaves is dependent on what the developer owned. Determining ownership and the related concept of developer expertise has received considerable attention in the software engineering literature. Previous studies summed each commit to a software artifact to determine a developer's ownership and expertise [20, 2]. This measure is appropriate for expertise assessment because each change increases a developer's knowledge. However, a commit based measure is not representative of the current state of knowledge in the system. Files and lines of code that have been deleted no longer need to be maintained. With a commit-based approach the deletion and addition of lines are counted equally. However, the deletion of a line removes knowledge that must be maintained, while an addition increases the maintenance burden. As a result, a commit-based approach is inappropriate when assessing the amount of knowledge that must be maintained when a developer leaves a project. For example, a team could delete the leaving developers module reducing the knowledge loss in the system to zero.

Instead of a commit-based approach, we use a blame-based approach. The blame function present in version control systems, in our case `git-blame`, are able to determine the person who last changed a line of code. `git-blame` tracks moved lines of code assigning them to the developer who wrote the code not the developer who moved it. In this way we are able to follow the ownership over time of each line of code in the system. We limit our analysis to source files only, for example, on Linux we only consider '.c' and '.h' files.

Migration commits: Each of the projects we studied migrated their system to a

new version control system. When this happens the history of the system is eliminated. In `git-blame` the developer who made the migration commit will get credit for writing every line in the system. For example, when Linux migrated to git in 2005, Torvalds added every line to git without including the previous history. Since `git-blame` attributes Torvalds as the author of every line, if he left the project we would have 100% knowledge loss.

This problem of false attribution can be solved by excluding the migration commits on all projects. By excluding this commit, our analysis only includes development that occurs after migration. Since the projects were already active with a community of developers before migration, the first period after the migration will see a huge number of "new" developers making changes to the project. To avoid this problem, we do not analyze the first two years of data that follow the migration commit to allow the development team to conduct substantial work before we begin our analysis.

Shared knowledge: We are interested in shared knowledge because the greater the sharing of knowledge the lower the risk of turnover [11]. Previous works did not consider shared knowledge, for example, Robles *et al.* examined the number of individual lines of code that are abandoned on a project [16]. We consider shared knowledge at the file level. Previous works on knowledge distribution considered all developers who had modified the file overestimating shared knowledge by considering transient developers. A developer who owns one line is considered equal to the developer who owns 1000 lines in the same file. Since open source project have many developers who make a single contribution we are only interested in developers who own at least 10%

of the lines of code in a file.

Files abandonment: We considered a file to be abandoned in a given period if there was no developers who have previously changed that file. The rate of abandoned files is obtained by dividing the total number of abandoned files in a given period by the total number of files in the project in the same period.

Succession matrices calculation: We want to know the current situation of developers knowledge about each other tasks. The git-blame relation and the 10% ownership rule cannot be used because nobody owns any LOC once a file is abandoned. However, we still can use past changes relationship based on commits. The tables we use to execute the matrices calculation (see Equation 1) are:

- Task Assignments (Developers per files matrix): represents in which files a developer have worked in the past. For instance, a developer A have worked 232 times on file X.

- Task Dependencies (Files per Files matrix): represents how many times the pair of files have been changed together as a part of a commit. For instance, file X with file Z have been commited together for 158 times. Files that have changed together as part of the same change request share logical dependencies [6].

The output table shows the distribution of how much knowledge developers should be aware about the files.

$$(Devs/Files) * (Files/Files) \tag{1}$$

Table 1: Project summaries

Project Name	Period	Total Files	Total Devs	Core Devs
Linux	2007-2012	46679	7913	697
Chrome	2009-2012	45713	911	175
Gimp	2007-2013	3714	165	12
V8	2009-2013	2579	78	25

Successors precision and recall: To validate the efficacy of our potential successors to abandoned files, we performed a precision and recall calculation (see Equation 2 and Equation 3). First, we based our calculation on the top 10 developers in the list of potential successors in a given period. After that, we observed in the next period if one of the top 10 would be the developer to assume the abandoned file. Our True Positive (TP) are the developers who assumed abandoned files and they were in the top 10 list. Our False Positive (FP) are the developers who assumed abandoned files and they were not in the top 10 list. Our False Negative (FN) are the developers who did not assume abandoned files and they were in the top 10 list.

$$PPV = TP/(TP + FP) \tag{2}$$

$$TPR = TP/(TP + FN) \tag{3}$$

2.2.1 Project Selection

We select the following projects: the Linux Kernel, the Gimp image manipulation program, the Chromium (Chrome) web-browser, and the V8 JavaScript engine. The

projects range in size of development team and product as well as the community that surrounds them. Table 8 shows basic size information about the project and the time periods we analyze. We define the core team using Mockus *et al.*'s measure of the number of developers who contributed 80% of the development work over a period of time. [1]

The Linux kernel has the largest development community associated with it. Over six year period we study, the core team consists of 697 developers. [2] It has also been extensively studied by empirical software engineering researchers [12, 30], so our findings regarding developer turnover will contribute to a growing understanding of how this project functions.

Google Chrome is especially interesting because it is a Google-lead project that has an open source software license. Chrome development is conducted in public but the practices it uses mirror those used by Google internally. Over a a four year period Chrome's core team consists of 175 developers.

Gimp was selected as a replication from the work by Robles *et al.* [16] who studied the historical file loss when a developer leaves the project. We also analyzed the two other projects they examined: Evolution and Nautilus. However, since these projects are similar in size to Gimp and the results were similar, we do not present them in the interest of keeping our figures less cluttered. Gimp has a relatively small core team of 12 developers. This core team size is similar to that found on the Apache project by Mockus *et al.* [18].

[1] Unless otherwise stated, a period is three months or a quarter

[2] Not all core developers are active at the same time

Ricca *et al.* [26] studied the knowledge distribution of a project when a group of developers leaves. However, as we will discuss later, their algorithm has a time complexity of $\binom{n}{g}$ making their calculations impractical on large projects. We selected three projects from their study to replicate in our own. However, these projects were so small, with only a few core developers, that the results are uninteresting. For example, the core team on the Closure compiler has only three core developers and Erlide has only one. We do not need any analysis to determine that if any one of these core developer leaves the project will be effectively abandoned. We present results only for the V8 project, which has 25 core developers.

2.3 Damage from Past Turnover

RQ1: What is the rate of developer turnover? How much knowledge loss has occurred as a result of turnover?

To gauge the rate of annual turnover on a project, we use the British institute of management definition [33]. Figure 2 shows the turnover rate for each project. Core developers are defined as the group of developers who wrote 80% of the system [18].

$$\frac{NumberOfLeaversInAYear}{(NumAtBeginning + NumAtEnd)/2} * 100 \tag{4}$$

The turnover results are presented in Table 2. In terms of core developers for Linux there are on average 37 joiners and 42 leavers with an average core team size of 524. The Chrome team has grown dramatically and there are on average 30

Table 2: Developers turnover from 2007-2013

	2007	2008	2009	2010	2011	2012	2013
USTurn.	17%	19%	16%	16%	14%	15%	15%
Linux	38%	44%	47%	49%	55%	69%	na
Chrome	na	na	24%	22%	34%	42%	na
Gimp	10%	25%	54%	49%	47%	47%	na
V8	na	na	42%	41%	79%	42%	31%

Table 3: Core developers turnover from 2007-2013

	2007	2008	2009	2010	2011	2012	2013
USTurn.	17%	19%	16%	16%	14%	15%	15%
Linux	4%	5%	5%	7%	10%	19%	na
Chrome	na	na	6%	6%	6%	11%	na
Gimp	0%	25%	0%	15%	18%	50%	na
V8	na	na	9%	7%	50%	43%	11%

joiners and 9 leavers with an average core team size of 134. The average turnover rate in the US for 2008 to 2013 is 15.2%, this estimate was performed by Compdata Surveys which includes data from over 34K companies in the US. [3] There are a large number of developers that contributed in only one year. The overall turnover ratio of the surround community fluctuates dramatically and is much higher than the US national average.

For Gimp and V8 that have 12 and 25 core developers respectively, we see periods of relatively little turnover followed by large turnover. The rates range from 9% to 50% for the projects and hover above the US average. The impact of transient contributors on these small projects is much more influential and we see a transient turnover rate of 10% to 54% for Gimp and 31% to 79% for V8.

How much knowledge loss has occurred as a result of turnover?

[3]Turnover information available at `http://www.compensationforce.com/2014/02/2013-turnover-rates-by-industry.html`

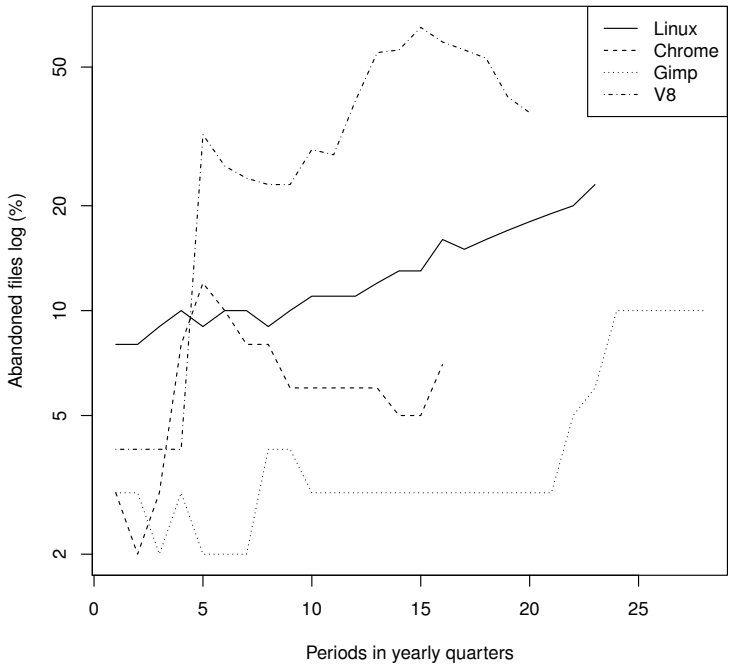

Figure 2.1: The percentage of files that are abandoned per period

We defined a file to be abandoned when there is no developer who owns at least 10% of the file. In Figure 2.1 we can see that file abandonment can be quite high. While some projects fluctuate dramatically, most notably, V8 follows an increasing pattern as developers write code and eventually leave the projects. For V8 on the last quarter, we observe a reduction in the rate of code abandonment. This increase in abandonment provides evidence for the decay of software systems [9]. Decay and entropy are inevitable even on an intangible software product as the most conscientious leaver who trains other developers will leave a gap in the teams understanding of the system.

2.4 Truck Factor

RQ2: How susceptible is a project to future turnover?

The 'truck factor' is essentially a measure of how the knowledge is spread across the development team. To calculate the truck factor, we measure the number of files that will be lost when a group of size g leaves.

Zazworka *et al.* manually calculated the truck factor to determine the distribution of development effort on Agile student projects [38]. Ricca *et al.* contributed a naive truck factor algorithm that in all cases has a time complexity of $\binom{n}{g}$ combinations [26]. As a result, they can only calculate the truck factor for small projects [34]. We contribute a stopping condition. We prove that the stopping condition allows us to calculate the truck factor for large projects.

These previous papers also have a significant flaw – they calculate the truck factor value at the most recent time point, but do not consider that some of the developers have already left the project. For example, a developer who left the project in 2005 will still be included in the developer combinations considered by Ricca *et al.* [26] when they calculate the truck factor in 2014. We exclude developers who have already left from our truck factor calculations.

2.4.1 Limitations of the Naive Algorithm

The naive truck factor algorithm proposed by Ricca *et al.* [26] has a prohibitively high time complexity in all cases. When n is the size of development team and m is

the total number of files in the project then the time complexity is given by:

$$T(n, m) = \sum_{i=1}^{n} \frac{n!}{i!(n-i)!} * m \tag{5}$$

For example, if a project has 30 developers then the number of developer combinations is over 17 million. This time complexity means that in their work they consider only small projects with the largest project having only 38 total developers[26, 34]. We are considering much larger projects, for example, Chrome and Linux have hundreds of contributors. It is impractical to compute all developer combination using this algorithm. Besides, even with improvements in the truck factor algorithm, we still could only obtain a group of 7 and 13 for Chrome and Linux, respectively.

We introduce a stopping condition which we prove identifies, for a given group size g, the set of developers who's loss will result in the maximum file loss.

2.4.2 Truck factor definitions

We define the following symbols:

D = the set of all developers on the project.

d_i = a particular developer in D, i is the id of the developer.

F = the set of all files in the project.

f_j =a particular file in F, j is the id of the file.

$M(f_j)$ = the set of developers who have modified the file f_j.

$I()$ is the logic function defined as:

$$I(condition) = \begin{cases} 1 & \text{if condition is true} \\ 0 & \text{otherwise} \end{cases}$$

We calculate the proportion of the file f_i that developer d_i owns:

$$L(d_i, f_j) = \frac{1}{|M(f_j)|} * I(d_i \in M(f_j)) \tag{6}$$

Then the file loss (FL) function returns how many files would be abandoned if a given developer combination C left the project:

$$FL(C) = \sum_{d_i \in C} \sum_{f_i \in F} (I(M(f_i) \subseteq C) * L(d_i, f_i)) \tag{7}$$

Then the maximum file loss for a given group size g is returned by the truck factor function:

$TF(C) = max(FL(C))$

We can see that a naive implementation of this algorithm will have a complexity of $\binom{n}{g}$, which is impractical to compute for large projects.

2.4.3 Stop Condition

The shared proportion of the number of files a developer d_i has modified on the project:

$$L(d_i) = \sum_{f_i \in F} L(d_i, f_j) \tag{8}$$

The upper bound of the file loss for a given developer combination C is:

$$UFL(C) = \sum_{d_i \in C} L(d_i) \tag{9}$$

Since

$$\forall condition : I(condition) \leq 1$$

We have:

$$FL(C) \leq \sum_{d_i \in C} \sum_{f_i \in F} [1 * L(d_i, f_i)] = UFL(C) \tag{10}$$

For each group size, we order the developer combinations by their UFL value and calculate the FL for each developer combination until the stopping condition, $UFL(C) < max(FL)$ is met.

In other words, we have shown that the number of files that a given group of developers modify, UFL, will be less than or equal to the number of files that these developers own exclusively, FL. Provided that we order the developers by the number of files that they modify, UFL, we can stop when the maximum loss that we calculated from the FL function is greater than or equal to the total number of files that the subsequent group of developers modify. The pseudocode for the **TruckFactor** algorithm is presented below.

Algorithm 1: Pseudocode for **TruckFactor** algorithm

Data: *developerList*, the list of all developer in the project sorted by their L value descending

$length()$, the length of the input list

g, the given combination size

$FL()$, the FL value for a given group of developers

$UFL()$, the UFL value for a given group of developers

Output: output all developer combinations that have the UFL larger than the stopping condition and their FL value

```
initialize an empty array: list
stop =FL(Combo(0, 0))
for i ← 0 to [length(developerList) − g] do
    if UFL(Combo(0, i)) ≥ stop then
        list.append((Combo(0, i), 1, i + 1))
    else
        break, the stopping condition has been met
while length(list) > 0 do
    remove the first element in list and assign it to e
    if e[1] <= g then
        for i ← e[2] to [length(developerList) − g + e[1] − 1] do
            if Combo(e[1], i, e[0]).UFL > stop then
                list.append(Combo(e[1], i, e[0]))
            else
                break, the stopping condition has been met
    else
        output combination e[0] and FL(e[0])
        update stop = Max(stop, FL(e[0]))
```

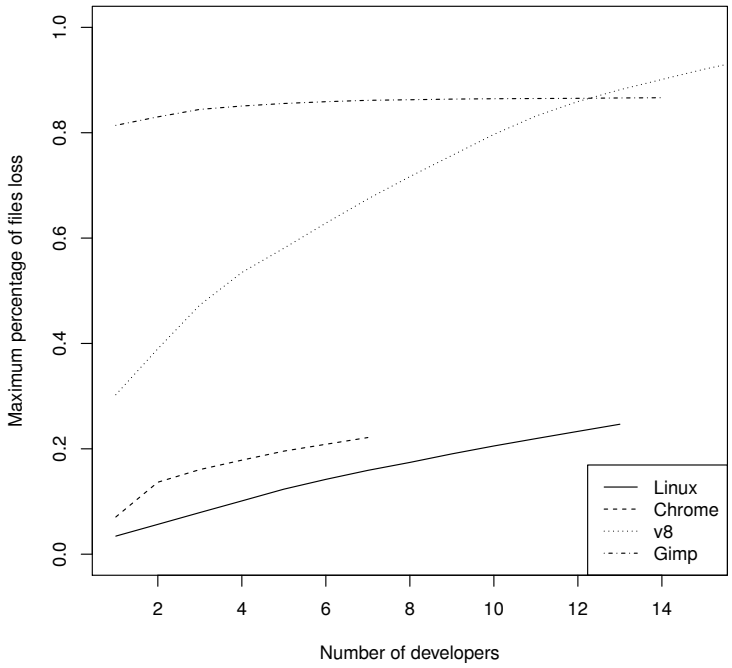

Figure 2.2: The percentage of files loss per number of developers

2.4.4 Knowledge loss

In Figure 2.2 we calculate the knowledge that would be lost on each project for a given group size. To obtain a reasonable comparison between the small group of core developers from GIMP and V8 against the larger group of core developers from Linux and Chrome we decided to work with a group of developers up to 20 developers.

On projects with a large core group size, the knowledge is distributed across a larger number of developers, see Linux and Chrome in the figure. In contrast, Gimp and V8 are highly dependent on a small number of developers. With Gimp the loss of a single developer would lead to the loss of 80% of the files in the system. For V8, we see a more even knowledge distribution, with one developer owning 30% of

the system. At this period, the V8 core consists of 9 developers, so a loss of 4 top developers would lead to a loss of only 50%. Although V8 had suffered from a high degree of turnover in the past (See Figure 2.1), for the size of the core team, the project has a reasonable knowledge distribution when compared to GIMP.

2.5 Successors

RQ3: Can we mitigate the impact of developer turnover by suggesting potential successors?

A development team can either hire a new developer or assign an existing developer to the abandoned files left behind by the leaver. We first ask, *how much experience does the developer who takes over maintenance have?*

There is a large literature on mentoring and integrating new developers into software projects. For example, Zhou and Mockus examined the impact of development environment on new developers [40]. Bird *et al.* [1] looked at the survival rate of new developers. Zhou and Mockus examined the amount of time until a developer becomes productive. Mockus [19] suggested mentors for developers based on past work and Canfora *et al.* [5] suggested mentors based on the email communication network.

While adding a new developer seems an obvious solution to file abandonment, on the projects we studied, the experts adopted these files.

In Figure 2.3 we found that median experience for developers to take over abandoned files is 1 year for Chrome, 2.75 years for Linux, 7 years for GIMP and 1.4 years

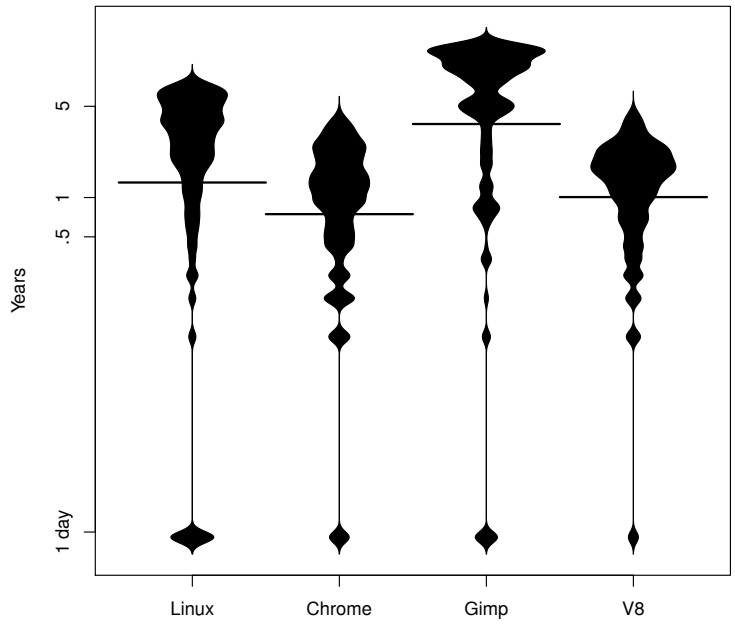

Figure 2.3: Experience of developer taking over maintenance of a file

for V8. It is clear that experienced developers tend to take over abandoned files. Developers who had less than six monthly of experience account for a small proportion of the adopted files: 9% for Gimp, 20% for V8, 17% for Linux, 29% for Chrome.

Our findings are consistent with Zhou and Mockus's [39] developer learning curve. They find that although the number of tasks a new developer takes on plateaus at three to 12 months, when the centrality (*i.e.* how many files are included in a task) and difficult are accounted for, developer productivity continues to increase over the entire measurement period of three years. In our study, we see that the most experienced developers tend to be the ones taking over the maintenance burden of abandoned files. Future work determining the type of files that newcomers tend to take on would

be interesting.

2.5.1 Possible Successors

How many possible successors are there?

For each abandoned file we want to determine which developer has the most expertise related to the file. We calculate the developer to file matrix (Dev-File) based on the number of times a developer has changed a file. We calculate the file to abandoned file matrix (File-AbandondedFile) based on the files that have co-changed with each abandoned file. We multiple the Dev-File matrix with the File-AbandondedFile matrix and are left with the Dev-AbandondedFile matrix. Since there are no developers who have changed the abandoned file left on the project, the Dev-AbandondedFile matrix represents the number of times each developer has changed a file that has co-changed with the abandoned file. We rank possible successors based by the number of files that they have changed that have co-changed with the abandoned file.

The intuition behind this succession measure is that if file A has changed with file B, and file A becomes abandoned, then the developer who works on file B will likely know something about file A. Our measure incorporates the commonly used measure of developer experience [20] and co-changes among files [4]. The measure can also be seen as incorporating the first matrix multiplication in Cataldo *et al.*'s [7] coordination requirements measure, adopting the assumption that a developer should know about files that changed with the files he or she works on.

On large projects like Linux it can become very computationally expensive to

perform these matrix multiplication. We also eliminated commits that contained over 100 files as Hindle and German [15] showed that these commit are misleading as they usually represent uninteresting changes, such as changing the copyright for all files in the system. The developer making this massive change is unlikely to understand all the relationships between these files. We implemented a database approach to multiply matrices and only include those files which co-changed with abandoned files making it possible to perform the multiplications on large projects.

In Figure 2.4, we find that the V8 team is very cohesive. We see that the median number of possible successors is 23 developers. Which means that the core team of developers is very aware of the entire system. Interviews of developers on the similarly sized projects, Apache and Subversion, indicated that developers felt comfortable with the entire codebase [29]. We see a similar phenomenon with V8. We also see this with the V8 truck factor that increases gradually with group size indicating a relatively uniform knowledge spread.

Over the long lifetime of the Gimp project there have been periods with a large number of potential successors. The median is 20. However, as of 2013 there are only three core developers. As we see in Table 2, Gimp has suffered from high level of developer turnover and currently has low knowledge spread leaving it susceptible to future turnover, see Figure 2.2.

For Chrome, the distribution is bimodal. We see that the bottom 50% of abandoned files are co-changed by a relatively small number of developers. However, there is an equally large group that has 100's of potential successors. Chrome clearly has

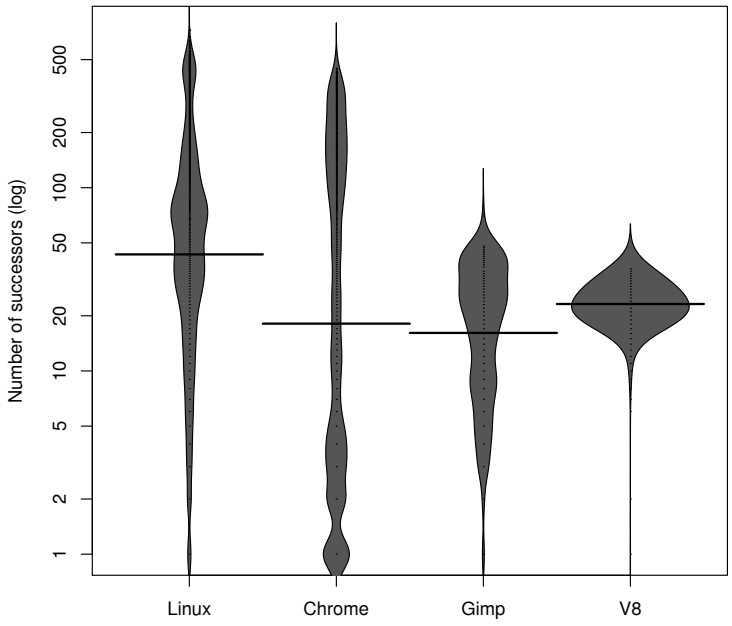

Figure 2.4: Number of Successors

modules that are at higher risk to turnover. One possible explanation is that Google pays developers to work on particular modules so that knowledge is less spread out. Future work could examine the importance of each file and determine if the level of co-ownership is high enough to mitigate the risk of turnover.

The Linux distribution is similar to V8 in that the knowledge is dispersed more evenly among the development team. Files that are abandoned have a median of 50 successors who have modified co-changing files. Although Linux is divided into different modules, such as networking vs USB support, the shared knowledge across each of these modules remains high reducing the risk of turnover.

2.5.2 Suggesting Successors

Can we mitigate the impact of developer turnover by suggesting potential successors?

In answering this question, the work that is most similar to ours is that of Mockus who created a measure to find mentors and successors for a developer who's job is being moved off-shore [17]. Much of the complexity of Mockus's measures relate to the need to determine which developer began modifying a file first to determine who is mentor and who is follower. In our work, the file is abandoned, so no current developer has modified the file. Furthermore, Mockus only considers the ownership matrix (*i.e.* Dev-File), while we multiply this matrix with the files that have co-changed with the abandoned file. Our output suggests who should take over an abandoned file, while Mockus's output suggests which developer should succeed or train a new developer.

The measure from the previous section calculates the potential successor of an abandoned file based on the files that have co-changed in the past. At each period, we use our succession measure to suggest the top 10 potential successors for each abandoned file. We calculate the precision and recall by comparing our prediction with the developers who actually take over the maintenance of an abandoned file. Since we cannot predict new developers as successors, we present results including and excluding newcomers.

Table 4 shows the precision and recall highly depend on the size of the development team, the distribution of knowledge, and the number of newcomers who join the project. For V8, the knowledge is spread across the core team. The precision is 31% and recall is 40%. These low numbers suggest that there is little code ownership

Table 4: Successors - precision and recall

Projects	New dev	Precision	Recall
Linux	No	52%	54%
	Yes	40%	42%
Chrome	No	59%	65%
	Yes	44%	52%
Gimp	No	99%	100%
	Yes	57%	57%
V8	No	45%	55%
	Yes	31%	40%

making it more difficult to suggest the top ten potential successors. However, on such a project, succession is not difficult because developers can work on large swaths of the system.

For Gimp, there are a large number of transient developers, so the prediction without newcomers is very high, almost 100%. However, when newcomers are included precision drops to 57%. Future work could use the parts of the system that have little co-ownership to suggest areas where newcomers should focus their effort to reduce the risk of turnover.

For Linux and Chrome, the precision of 40% and 44% is impressive given that they have 100's of developers who have made significant contributions to the project. These values also indicate that the modularity of the projects is quite high and that subteams work in relative isolation.

2.6 Threats to Validity

The *generalizability* of our study is threatened by the use of open source data only. We have tried to mitigate this problem by also studying Chrome, which is a Google-lead project that employs Google developers and uses development practices that mirror those used internally at Google. The projects we have selected range by application type, such as infrastructure vs end-user, and inter terms of team and code as, the data for each project is publicly available. Large projects that are divided into modules could be viewed as separate smaller projects. Future work done at different granularities will indicate the risk of turnover at module level.

Ownership and dates: In the methodology section, we describe how we deal with the limitation surrounding our data collection and our calculations of ownership. Since there are many transient developers on open source projects, we used Mockus *et al.*'s definition of core group. Changes in this definition could lead to slightly different results.

Types of knowledge: We have only considered source files. Knowledge exists in documentation and in developer communication, so our quantification of source knowledge represent an important but limited view of all possible system knowledge. Future work could include creating similar measures to suggest knowledge loss from other software artifacts.

2.7 Contributions and Conclusion

We reported results related to our research question:

- the damage done to projects from turnover
- the susceptibility of project to future turnover
- a metric to mitigate turnover by suggesting possible successors based on files that co-changed with abandoned files.

The turnover rates among the core development team grows over time and ultimately matches that of the US average. On small projects, the turnover rate can be fluctuate drastically because the team size is small making the system highly susceptible to turnover. Further, we see that successful projects can maintain a relatively high level of file abandonment and continue to function effectively. It is possible that some of these abandoned files are related to legacy code that runs infrequently. Future work to determine the level of importance of abandonment would be interesting.

Past work on the knowledge distribution of a project or the 'truck factor' was hampered by the number of developer calculations that must be done, in all cases $\binom{n}{g}$ combinations. This limited the size of the development team on which the calculations could be run to small projects. We contribute and prove that our stopping condition finds the optimal solution and given the distribution of developer contributions, does so with relatively few combinations. This allows to perform the knowledge distribution and loss calculation on large projects, such as Linux and Chrome that

have hundreds of core developers. We hope that researchers and practitioners will use our implementation to calculate the potential knowledge loss on their projects.

In terms of mitigating the risks of turnover, we discover that successors tend not to be newcomers, but are experienced developers that have been with the project for multiple years. This result agrees with Zhou and Mockus's developer learning curve which indicates that developers continue to increase their knowledge even after having been with the project for multiple years.

We suggest possibles successor by combining measures of past developer expertise [20] and files that change together [4]. The novelty of our measure comes in examine files that co-change only with abandoned files and suggest successors for these files based on relevant developer expertise. We attain a reasonably high precision and recall. We discuss how the precision and recall is effected by the team size as well as the knowledge distribution.

Our results are relevant for both researchers and practitioners as they show how turnover, knowledge distribution, and team size influence software projects.

Chapter 3

Organizational volatility and post-release defects: A replication case study using data from Google Chrome

In the previous chapter, we discussed about the damage on knowledge loss caused by turnover, we calculated potential loss using truck factor algorithm, and we offered a list of possible candidates to assume abandoned code. In this chapter, we study the impact of changes in the development team, *i.e.* the arrival and the departure of employees in a project, on the number of post-release defects.

As developers join and leave a project, the organization of that project is effected. For example, new developers might bring innovations while departing developers will

leave knowledge gaps that other team members will have to fill. Our goal is to understand how changes to the development team affect software quality. Mockus [19] conducted a study on a single project at Avaya and determined that developers leaving the project had a small negative impact on software projects. In contrast, newcomers did not have a statistically significant impact on quality.

Our goal is to replicate Mockus's study in the context of Google Chrome. During Mockus's study period, Avaya was going through turnover in the form of layoffs and outsourcing[17]. In contrast, Chrome has seen drastic growth in terms of popularity and number of developers. This contrasting replication allows us to determine whether Mockus's findings generalize and to understand which variables are important in predicting organizational volatility in a growth context.

We use the same response variable as Mockus – the number of post-release customer found defects (CFD). However, we make two changes: first, we group measures at the directory level instead of the file level because few files have multiple defects, second, we use a count of CFDs instead of a binary response.

3.1 Quality Predictors

Many studies have modeled bugs, so we included control variables to take into account known strong predictors such as developer expertise as well as our leaver and joiner measures. Table 5 lists the predictors used in our model and in Mockus's model as well as a description of how each is calculated. Below, we provide a brief conjecture

Table 5: Quality predictors from Mockus's paper. We are unable to include certain predictors based on the available Chrome data. All predictors are measured per release at the directory level.

Predictors	Included	Description
Co-owners	Yes	The number of developers who have worked in a directory. Mockus calls this "size of organization"
Time from prior and next change	No	Mockus had information about changes to the development team. We do not have this information for Chrome.
Leavers	Yes	The number of developers who left the project per directory
Joiners and Newcomers	Yes	The number of developers who joined the project per directory
Churn	Yes	The sum of the lines of code added and deleted from all files in a directory.
Co-changing directories	Yes	The number of directories that have co-changed with a directory. Mockus calls this "logical dependencies." The greater the number of directories in a commit the greater the complexity of the change.
Change Diffusion	Yes	The maximum number of co-changing directories in a commit. We drop change diffusion from our model as it is highly correlated with the number of co-changing directories.
Release Dependencies	No	At Avaya, a change could become part of multiple releases, for Chrome, each change is included in only one release.
Workflow	No	We do not calculate the developer workflow network
Developer Experience	Yes	The minimum number of years of experience across all developers working in a directory.
Distributed Development	No	We are unable to count the number of different offices Chrome developers work in.
Mentor Offshore	No	We do not calculate mentors and cannot know when a developer is working offshore.

on the influence of each predictor.

Churn: The more changes are made to a software artifact the more defects that are found in it. Larger size generally correlates with complexity and increased bug density [13, 24].

Co-owners: There is evidence that when more developers touch a software artifact there will be more bugs [2, 19].

Leavers: When a developer leaves a project the team losses that developer's knowledge of the system. This can lead to tacit knowledge gaps that can result in defects [19].

Newcomers or Joiners: Newcomers may not be experienced on the project, but they can bring fresh ideas [8]. Since the core team of Chrome are Google employees, they are stringently vetted increasing the likelihood that they will be good additions. Newcomers may, however, introduce defects through a lack of knowledge about the system and will take time from core developers when they ask questions.

Co-changes and change diffusion: The greater the number of co-changing software artifacts the lower the quality of a software is expected to be. The idea of logical coupling was introduced by Gall *et al.*'s [10] who implied that logical coupling relates to the files that change together in a commit. Change diffusion is the maximum number of co-changing directories in a commit. Mockus *et al.*'s [21] found that change diffusion was an important predictor to estimate defects.

Developer experience: The quality of a system is tightly coupled to the experience and expertise of the developers. Further, experienced developers have a detailed

understanding of the design and evolution of a system and are less likely to introduce defects [22].

The chapter is structured as follows. In Section 3.2, we describe the methodology and data. In Section 3.3, we present and discuss our results. Finally, we present conclude the chapter.

3.2 Methodology and data

3.2.1 Data

Google Chrome was started in 2008. We mined Chrome data from July 2008 to May 2013. It is developed by more than 1000 contributors and it has more than 150K files. A core team of 176 developers have made 80% of the changes to Chrome. Since our goal is to predict the number of post-release defects, we describe the Chrome release process and how we partition each change and bug into a release.

Chrome uses the issue tracker provide by Google Code.[1] Issues include a summary, a detailed description of the issue, and attachments. After an issue is opened, there is a section where we can find specific details such as status, owner, type, priority, release, operational system and release block. We are able to differentiate between bugs and feature requests, and only include bugs in this work. We are able to link bugs to commits, because the developer who fixed the bug records the commit number. The identification of CFDs is important to our prediction model because all commits

[1]Chrome issues website `http://code.google.com/p/chromium/issues/list`

have an issue number. This fact would misrepresent our data, given that commits would be highly related to issues.

Classify bugs as CFDs: First, we use the information we mined from the bug database. Second, we need to classify which of these bugs are opened by users. The issue form has a field called "Reported by". In this field, we can find the reporter's e-mail and determine whether the reporter is a project member. If the creator is not a project member, then the bug was opened by a user.

Releases identification: The official release dates for Chrome is reported in Chrome developers web page.[2] Starting at release 5 Chrome transitioned to a rapid release cycle and produces a release every 6 weeks. We use Rahman and Rigby's tool to extract and differentiate development changes from post-release changes [25].

Developers identification: The developers are identified by the email addresses they used to submit the commit. There are some duplicated email addresses. To resolve this issue, we use Canfora *et al.* [5] name aliasing tool. However, the tool had some flaws. For instance, we created a new automatic script to resolve names that had less than 5 characters.

Newcomers and people leaving the organization: For a given developer, we consider the date of his first change to be his starting date, and the date of his last change to be his ending date.

[2]Chrome release information available at `http://www.chromium.org/developers/calendar`

3.2.2 Methodology

Quality predictors

Mockus *et al.* [19] used release dates as the starting point for looking one year to the past at file level to calculate the factors mentioned on Table 5, instead of LOCs we use directory churn, which is a strongly correlated measure [13]. As a response, Mockus looked one year to the future, trying to identify if a file had at least one CFD for the specific file during the measurement period.

Our model will be different regarding the granularity level. Instead of examining files, we will use directory level. The Chrome release process is also more frequent, so we will consider the cycle of development channel (six weeks) for each release to calculate the predictors in Table 5.

Model selection

Mockus modelled bugs using a logistic regression, either a file had a bug or it did not. We tried the same model, however, the results were not significant because most files do not have bugs leading to a zero-inflated dataset. As result, we grouped bugs at the directory level and used bug counts as the response, instead of binary response. We considered poisson, zero-inflated poisson and quasi-poisson models to fit our data. We dropped the poisson model because there is overdispersion in our data. The zero inflated poisson model was compared with quasi-poisson model using the Vuong test available in R statistical software. The Vuong test selects the better model based on the likelihood ratio and non-nested hypothesis [35]. After testing both models, the

quasi-poisson model was superior.

3.3 Results and Discussion

There is an extensive literature on bug prediction models, so before we create a model with all of our predictors, we evaluate how well churn and the number of co-owners predict post-release defects. These two initial models give us a baseline against which to compare our future models. Our first model only includes the level of activity, *i.e.* the churn, that a directory has undergone. In Table 6 we can see that this simple model does quite well explaining 45.23% of the deviance. Other researchers have shown that churn is a good predictor of the number of defects in a module [24, 41]. We replicate these findings showing that the more changes that are made to a directory, the more post-release defects it will contain.

In our second model, we used the number of developers who work in a directory, *i.e.* the number of co-owners. We find that this model does even better explaining 66% of the deviance. Our findings support previous research that has shown that the greater the number developers working on a software artifact, the greater the number of defects [19, 2].

In our third model, the number of co-owners of a directory is highly correlated with all other predictors. This correlation means that the more developers who touch a file, the more churn, the greater the number of leavers, the greater the number of joiners, and so on. For this reason, we decided to normalize the other predictors by

co-owners. By dividing by co-owners, we are in effect taking the average per developer within each module. The resulting model, Model 3 in Table 6, explains 71% of the deviance. All predictors are statistically significant with the exception of the number of co-changing directories, which is dropped from further analysis.

Table 6: Prediction Models for CFDs

	Model 1	Model 2	Model 3 [b]
Churn	0.67		0.16
Co-owners		1.28	1.05
Leavers			-3.04
Newcomers			-0.62
Developer Experience			-1.60
Co-changes			*
Deviance Explained	45.23	66	71

[a] ($p<0.001$, except * $p<1$)
[b] Predictors normalized by co-owners.

Table 7: Effect of Organization, Directory, Change and Social factors on the Number of CFDs

	Variables	Estimate	10%	50%	100%	200%
Directory Churn	log(churn/co-owners)	0.16	1.53	6.67 [b]	11.66	19.10
Organization	log(co-owners)	1.06	10.63	53.69	108.48	220.42
	log(leavers/co-owners)	-3.04	-25.18	-70.89	-87.88	-96.47
	log(joiners/co-owners)	-0.62	-5.73	-22.20	-34.90	-49.35
Experience	log(experience/co-owners)	-1.60	-14.18	-47.81	-67.10	-82.83

[a] The quasi-Poisson dispersion parameter is taken to be 6.4 (over-dispersion). To make interpretation easier the proportional change at 10% to 200% is shown for each variable.
[b] For example, a doubling in the average churn made by a developer increases the number of post-release defects by 6.67%.

The interpretation of each variable is complicated because a quasi-poisson model has a log-link function and the explanatory model is on log scale. For these reasons, we investigate the rates of change shown in Table 7. For instance, a 50% increase in number of co-owners leads to an increase of 54% in the number of post release defects.

Directory churn: Normalizing churn by number of co-owners means that we are modelling the average churn of developers per directory. By tripling the average churn a developer makes, the number of post-release defects increase by 19%.

Co-owners: Increasing the number of co-owners in a directory has a high impact in the CFDs post-release. For instance, a doubling in the number of co-owners in a directory doubles the number of post-release CFDs.

This result adds to the growing consensus that too many developers touching a software artifact can introduce defects in the form of unexpected dependencies [6].

Number of developers leaving the organization: The correlation between the number of leavers and CFDs is positive. However, this relationship is largely influenced by the number of co-owners – the larger the total number of developers, the larger the number of developers who can leave. As a result, we normalized leavers by the number of co-owners, leavers/co-owners. With this normalization, we see that the more people who leave the fewer number of post-release defects reported by customers. This finding is unintuitive and the opposite direct to what Mockus found. We have conjectured that leavers would lead to knowledge loss and to less experienced team members taking over code they are unfamiliar with.

We suggest two possible explanations that deserve future work. First, when a developer leaves the project, the features they were working on may be put on hold, which would lead to fewer changes and fewer defects. Second, existing co-owners may take over the leaving developers work leading to a more focused set of changes and fewer defects.

Number of newcomers: Increasing the ratio of newcomers/co-owners in a directory decreases the number of CFDs post-releases. By doubling the ratio of newcomers, we see a decrease of 34.9% CFDs post-release. In Mockus's work, the impact of newcomers was not statistically significant. We find preliminary support that newcomers bring positive innovation without increasing the number of defects. One contextual factor related to Chrome is that Google's hiring process is very stringent allowing them to hire highly skilled developers. The impact of newcomers who are less stringently vetted may lead to different results.

Experience: The longer a developer has been on the project, the fewer the number of CFDs that are found in his or her code. For example, doubling the developer experience decreases the CFDs post-release by 27.31%. The more experienced people are in the project, the better the quality of the code is [22].

Co-changing directories: The number of directories that where co-changed did not have a statistically significant impact on the number of post-release bugs. This runs counter to the findings of Mockus and other who found that logical dependencies were important in determining post-release defects [19, 32]. Logical dependencies are usually calculated at the file level and the more coarse grained directory used in this study may have influenced the result.

3.4 Threats to Validity

This study is a replication of Mockus's [19] study of a switching system at Avaya. The context of our study is drastically different from the outsourcing that existed at the time of Mockus's study. As a result, some of our findings differ from those of Mockus and future replications are necessary to provide a more generalized understanding of organizational volatility and turnover. We were able to replicate most of the measures used by Mockus and feel that other should be easily able to replicate our work. Future work may also want to investigate different levels of granularity, such as directories vs modules.

Concerning the internal validity of our measures, the greatest threat is the reliability of bug data. The Chrome data set is an excellent source because we are able to differentiate internal issues from bugs reported by end users. Internal issues are highly related to churn, while end user bugs tend to be more representative of defects. One difficulty for further replication will be finding projects that allow research to differentiate internal and external bugs.

3.5 Conclusion

In this chapter, we presented a contrasting replication of Mockus's research on organizational volatility [19]. We mined the data from Google Chrome project which is growing in team size and popularity and contrasts with the project at Avaya that was experiencing outsourcing.

We used three predictors that have been extensively studied. We found that the number of co-owners in a directory increases the number of CFDs found post-release. This finding adds to the growing consensus that as more developers work on a software artifact there will be more uncoordinated and buggy changes. In terms of churn, *i.e.* development activity, we found that higher churn leads to more complexity and greater post-release defects. We also found that the greater the developer experience, the fewer the number of CFDs post-releases.

After normalizing for the number of co-owners, we found that the greater the ratio of leavers in a directory, the fewer the number of post-release defects. This result is counter-intuitive and needs future work to determine if other factors are at play. We also found that adding new developers to a directory actually reduced the number of post-release defects. While we find this result surprising and deserving of future work, we suspect that Google does a good job of vetting replacement developers.

Our findings add another data point in our understanding of organizational volatility. There is a clear need for other studies in new contexts to strengthen our understanding turnover on software projects.

Chapter 4

A preliminary examination of dormant files on software projects

In the previous chapters, we discussed about the turnover and its impact on the loss of knowledge on software projects. We also described quality predictors, which included leaving and joining developers in a software project, and their impact in the quality of the software, we conclude that in Chrome's case the greater the number of leaving and joining developers the lower the number of defects. This findings deserve a future research. In this chapter we examine the impact of dormant files on software projects and we discuss the knowledge that is lost when developers stop working on a file.

The forgetting curve suggests that if students do not review information regularly, one month later they will retain only 20% of the original information and only 15% of the information after two months [31]. Applied to software development, this implies that code that is not regularly maintained will be forgotten even by the developer

who created it. We define a dormant file to be a file that has not been modified for at least one year. Maintenance of dormant files is difficult because developers no longer understand the file and its relationships to other files in the system. Without proper knowledge of a dormant file, developers can unintentionally make changes to related files that could lead to surprise bugs in unchanged files [32].

Since larger systems are known to be more complex [13], the accumulation of dormant files make maintenance more difficult. Further, when a newcomer joins a project, they not only need to understand active files, but also the dead weight dormant files.

4.1 Research Questions

The goal of this work is to provide a preliminary quantification of dormant files through the following research questions.

RQ1 *Total number of dormant files and LOC:* How many files are dormant and how many LOCs are contained in dormant files?

Over time, a system can become more complex and hard to understand. If files are not modified for a long period of time, it will be difficult for future developers to understand the untouched files. LOCs are highly correlated with source code complexity [13].

RQ2 *Dormant file lifecycle:* When will a file become dormant? How many dormant files become active and how many of them remain dormant? How long does it

take for a dormant file to become active again?

We want to understand the lifecycle of a dormant file. We first measure the amount of time from when a file was created until it became dormant. We also measure the time between the changes that happened immediately before a file becomes dormant to understand whether a file suddenly or gradually becomes dormant. Once a file is dormant, we measure how long it remains dormant. Files that never become active again are excluded from this last measurement.

RQ3 *Experience:* How much experience does a developer who works on a dormant file have?

We conjecture that the greater a developer's experience on a project, the easier it will be for the developer to understand a dormant file. Experienced developers can relate their knowledge of the project with that of a dormant file. We compare the experience of developers that take over dormant files with the experience of developers of active files.

RQ4 *Surprise Bugs:* How many dormant files have surprise bugs?

A bug that is found in a dormant file will have been present for at least a year. These bugs are a surprise as they have remained untriggered for an extended period of time. It is also possible that changes related to the dormant file may have triggered the surprise bug by, for example, causing the dormant file to be used in an unexpected manner [32].

The remainder of this paper is structured as follows. In Section 4.2, we describe our methodology and data. In Section 4.3, we present the results. In Section 4.4, we

Table 8: Project summaries

Project	Period	Files	LOC	Devs
Chrome	2008-2013	32481	23595185	1205
Evolution	1999-2014	1681	201255	524

describe the threats to validity. In the final section, we conclude the paper.

4.2 Methodology and data

4.2.1 Project Selection

We study Google Chrome and Evolution. The projects respective sizes and time periods are in Table 8. Chrome is a Google-lead project and uses similar development processes to those used internally by Google. Chrome has more than a thousand contributors. Evolution was selected because it is an example of a legacy system [3]. Evolution has a relatively small core team of 24 people while Chrome has more than 100 people in its core team.

4.2.2 Method

Data extraction: We mine the data from Chromium and Evolution from their respective git repositories. We consider only the .c and .h source files.

LOC: To obtain the LOC, we use a script that counts the LOCs per file.

Dormant files identification: First, we obtain the history of file changes (commits). Second, we ordered each commit in ascending order by its date. Third, we calculate the difference between the dates each consecutive commit. Fourth, we mark the files

related to commits where the differences are greater than one year as dormant files.

A file's least active period before it becomes dormant: We use the commit date as above, but from each of the selected commits, we consider the previous two commits. We calculate the difference between these dates to give a sense of how active the code was before the current commit.

Dormant file becoming active: Once a file has been marked as dormant, we look for future commits and calculate how long it takes until the dormant file becomes active. If no future change is made to a file, it remains dormant and is excluded.

Experience: We consider the project experience of a developer in years as the difference between the authors first commit and the date the he or she modifies a dormant file.

Bug extraction: To identify bugs on the Evolution project, we search for "bug" in the commit log. The files changed in this commit are considered to be related a bug fix. For Chrome, we use the issues that are reported in the project's issue tracker. We only take issues that are tagged as a bug fix and that are linked to a commit, so that we are able to identify the files involved in bug fixes.

4.3 Results and Discussion

RQ1 *Total number of dormant files and LOC:* How many files are dormant and how many LOCs are contained in dormant files?

We observe in Figure 4.1 that in Evolution, of the total 1681 files, 49% are dormant

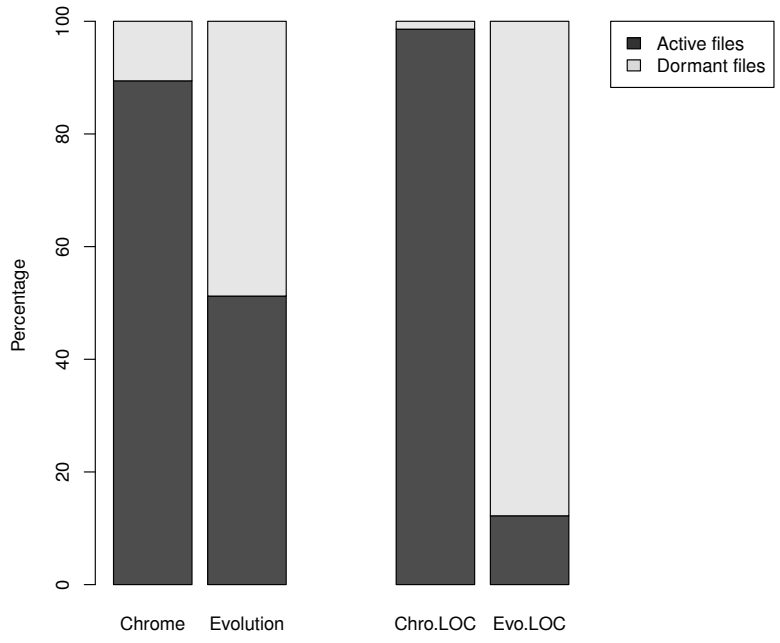

Figure 4.1: Total of dormant and non-dormant files and its LOC

and 51% are active. In terms of LOCs, for Evolution, there are a total of 201 KLOC, 88% belongs to dormant files and 12% belongs to active files. This level of dormant files makes it clear that Evolution is a legacy system. With more than 15 years of history, Evolution is now evolving at a slow pace and the dormant files are responsible for a great part of the system and its complexity.

For Chrome, we find that, of the total 33,060 files, 10% are dormant and 90% are active. In terms of LOCs in Chrome there are a total 23.6 MLOC, 1.4% belongs to dormant files and 98.6% belongs to active files. Chrome is a younger project when compared with Evolution. Chrome has a remarkably small number of dormant files indicating that the project is constantly evolving and growing. Chrome's dormant

files represent few LOC compared to the entire system. In other words, the dormant files are not adding much complexity to the project.

RQ2 *Dormant file lifecycle:* When will a file become dormant? How many dormant files become active and how many of them remain dormant? How long does it take for a dormant file to become active?

The amount of time from when a file was created until it became dormant is 1479 days for Evolution and 609 days for Chrome.

In Figure 4.2, we observe that for Chrome and Evolution the period before a file becomes dormant is 39 and 106 days, respectively. These changes that proceed dormancy indicate that when a file has not been changed in the last 2 to 4 months, it is more likely to become dormant.

For Evolution, we find that, of the total 820 dormant files, 52% become active from dormancy and 48% remain dormant. For Chrome, we find that, of the total 3496 dormant files, 49% become active from dormancy and 51% remain dormant. The rates of becoming active from dormancy and remaining dormant are fairly constant across the two projects.

The period of time for a dormant file to become active is 457 days for Chrome and 623 days for Evolution. This time gap is problematic as developers will likely remember little of the content of a dormant file. For example, the forgetting curve suggests that if we do not review information regularly after one month we will only retain 20% of the original information [31]. While future work is necessary, the dormant files that have not been touched by anyone for more than one year will be

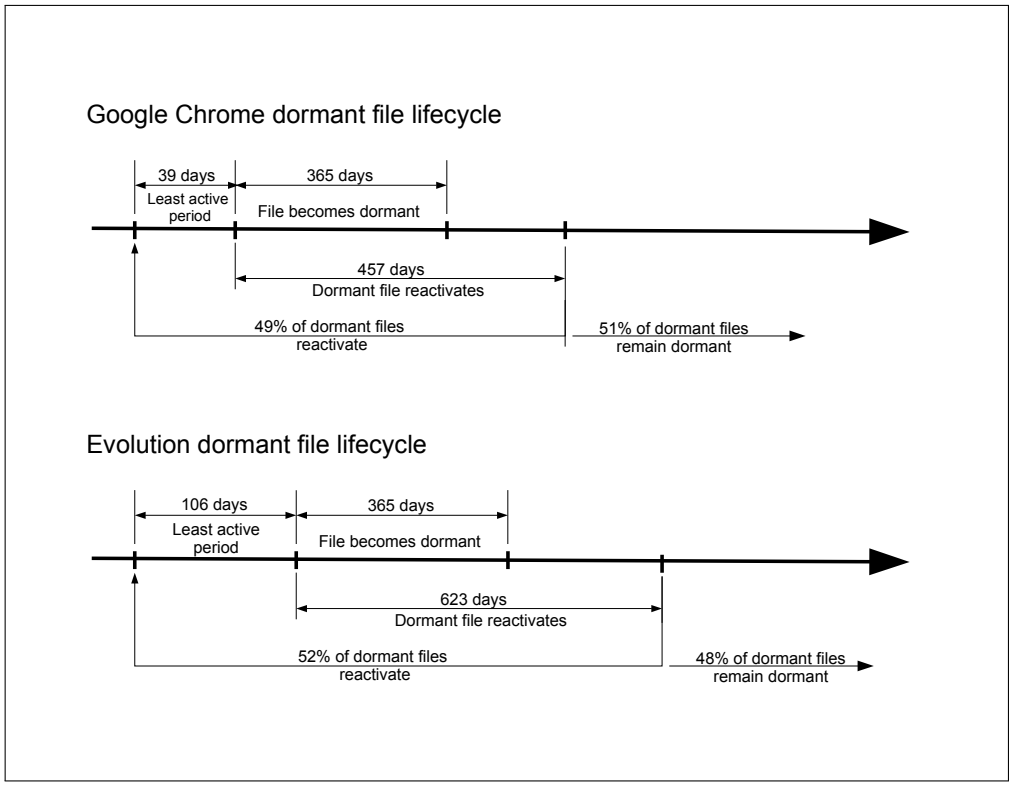

Figure 4.2: Dormant file lifecycle

mostly forgotten. We suspect that overall expertise on the system may increase a developers ability to recall a file.

RQ3 *Experience:* How much experience does a developer who works on a dormant file have?

Figure 4.3 shows the distribution of the experience for developers who modify dormant files and those who modify active files. In Chrome, we find that the median experience is 1.12 years for developer of dormant files and 1.2 years for active files. A Wilcoxon test shows that this difference is statistically significant with $p << 0.001$. The median developer experience for Chrome is almost similar for both conditions, however, it is clear that developers working in both groups tend to be experienced.

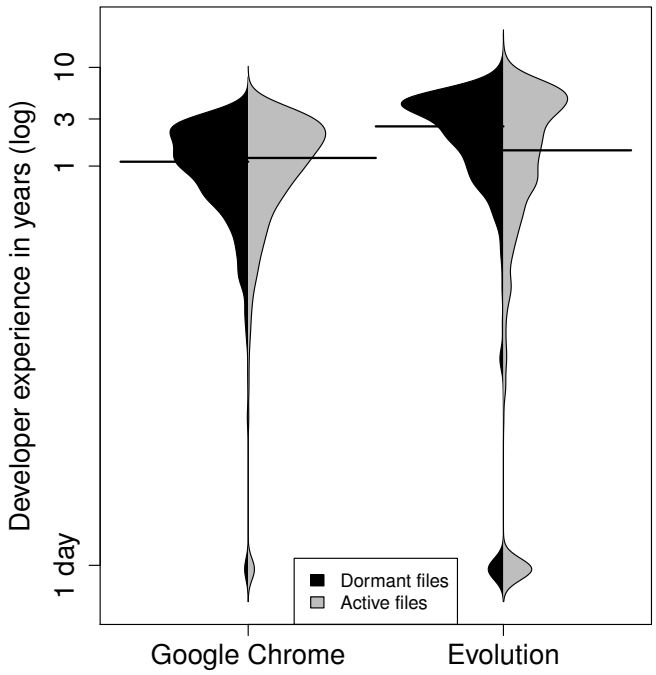

Figure 4.3: Experience of developers who assumes dormant files

In Evolution, we find that the median experience is 2.85 years for developers of dormant files and 1.44 years for active files. A Wilcoxon test shows that this difference is statistically significant with $p << 0.001$. The median developer experience for Evolution is greater for developers who work on dormant files. This experience may alleviate the negative impact of the forgetting curve and the complexity of large dormant files that exist on the system.

RQ4 *Surprise Bugs:* How many dormant files have bugs?

The proportion of bugs found in dormant files on Chrome and Evolution is 5% and 17%, respectively. The proportion of bugs found in active files on Chrome and Evolution is 23% and 19%, respectively.

Dormant files have lower chance to have bugs in Chrome and a almost equal chance in Evolution. Chrome is a younger project and we observe in Figure 4.1 that most of it is active. Evolution is an old project and half of its code is active and half is dormant. Bugs in dormant files can catch developers by surprise and may be more difficult to fix as the knowledge of these files is lost.

4.4 Threats to Validity

Our findings may not generalize outside the two projects that we studied. However, the projects were selected to contrast, with Chrome being the rapidly evolving system and Evolution being the legacy system.

We mined bugs from the commit log. An email to a core developer on Evolution confirmed that they require committers to link each bug to the commit that fixed it. Chrome follows a similar practice requiring each change to be linked to an issue. This bug linking technique has been widely used in the MSR community.

Our measures of dormant files are preliminary and based on an arbitrary one year time frame. Given that over 20% of knowledge is forgotten by students after one year [31], we feel that developers will have little recollection of changes made over a year ago. Future work, however, could experiment with other values and may find that even more of the system is dormant than we measured. Alternatively, developers may modify files related to a dormant file, thereby maintaining their knowledge of the unchanged file, in this scenario we would be overestimating knowledge loss.

4.5 Conclusion

Dormant files make a system harder to understand and more difficult to maintain. In this preliminary work, we have determined that a large proportion of Evolution, a legacy system, is dormant, 88% of the LOCs and 49% of the files. In contrast, only a small proportion of the actively growing Chrome project is dormant, 1.4% of the LOCs and 10% of the files. Of the files that become dormant, on both project approximately half of the will become active again with in the next 1.5 to 2 years. Future work is necessary to understand what drives these changes.

Surprise bugs are contained in files that do not change. For Chrome, we found that dormant files had fewer bugs than active files, but that 5% of the dormant files contained a dormant bug. For Evolution, bugs are found in equal proportion in active and dormant files. Evolution clearly suffers from a stale codebase. Interestingly, new developers do not take over dormant files, instead developers with a median of at least one year of experience make changes to dormant files. These developers likely have a better understanding of the interactions of the system and so are more qualified to understand and modify dormant files. Future work is necessary to study the activity patterns of dormant files and to create models of dormancy. Future work is needed to identify the proportion of the dormant files that are only inactive and the proportion that are stable code. Future work is needed to evaluate how beneficial is the experience of developers who assume dormant files. Future work is necessary to evaluate if the surprise bugs are actually bugs and not just a ripple due to feature or

requirement changes.

Chapter 5

Conclusions

The final chapter, we conclude this work with a discussion of the impact of knowledge loss on software projects caused by turnover, customer found defects and dormant files. Section 5.1, is described in the context of what we found in our previous chapters, derived from different OSS software empirical observations and methodology. Section 5.2 future work that was derived from the research we performed.

5.1 Contribution of the empirical study on the impact of knowledge loss on software project caused by turnover, post-release defects and dormant files

The main goal of this work is a contribution of the impact of knowledge loss on software project caused by turnover, post-release defects, and dormant files. Below we revisit the previous chapters, their research questions, data, and methodology. Finally, we present our contribution combined with our findings.

Turnover and succession (Chapter 2): By examining four OSS, Linux, Google Chrome, GIMP, and V8, we were able to measure turnover, the loss of knowledge and to define skilled people in the project. We addressed the following topics in our research questions:

- the damage caused to projects from turnover
- the susceptibility of the project to future turnover
- a metric to mitigate turnover by suggesting possible successors based on files that co-changed with abandoned files.

Organization volatility and defects (Chapter 3): We replicate an empirical study using Google Chrome project. We were able to create a statistical model of customer-found defects. We addressed the following quality predictors:

- Churn,
- Co-owners,
- Leavers,
- Newcomers or joiners,
- Co-changes and change diffusion.

Dormant files (Chapter 4): We performed a preliminary examination of dormant files in software projects. We addressed the following topics:

- Total number of dormant files and LOC,
- the lifecycle of a dormant file,
- the experience of developers who assumes a dormant file,
- and the proportion of dormant files that presented defects.

Knowledge spread:

We observed that the loss of core developers grows over time. Small projects depend heavily on core developers. When core developers leave small projects, they take all the knowledge about the system and cause the project to fail. In contrast, larger projects have knowledge spread among developers. For this reason, larger projects are more likely to continue for a long time than smaller projects.

Abandonded code:

We discover that successors tend not to be newcomers but are experienced developers that have been with the project for multiple years. This result agrees with Zhou and Mockus's developer learning curve which indicates that developers continue to increase their knowledge even after having been with the project for multiple years.

Defect prediction model:

Using Google Chrome, we found that the greater the ratio of leavers in a directory, the fewer the number of post-release defects. This result is counter-intuitive and needs future work to determine if other factors are at play. We also found that adding new developers to a directory actually reduced the number of post-release defects. While we find this result surprising and deserving of future work, we suspect that Google does a good job of vetting replacement developers.

Knowledge decay:

We observed that in newer systems such as Google Chrome, it code base is highly active. In contrast, a legacy system as Evolution OSS, most part of its code is dormant. For this reason, the longer it takes to developers to change these files the less will be remembered, as already reported in previous research about forgetting curve [31]. Interestingly, new developers do not take over dormant files, instead developers with a median of at least one year of experience make changes to dormant files. These developers likely have a better understanding of the interactions of the system and so are more qualified to understand and modify dormant files.

5.2 Future Work

In the previous section, we discussed the contribution of the impact of knowledge loss on in software projects. In this section, we state the future works that emerged from the research we performed. They will be presented in the form of subsections with the chapters names.

5.2.1 Developer Turnover and Succession: Empirical studies of the susceptibility and damage from developer turnover

The future work that emerged from the research on the Developer Turnover and Succession are the following:

- Analyzing and suggesting areas of the system with little or no ownership to newcomers
- determining the type of files that newcomers is likely take on
- examining the importance of each file and determining the optimal degree of co-ownership to mitigate the risk of turnover
- performing an analysis of turnover, truck factor and succession at different granularities, e.g. module level, instead of the file level
- replicating the same methodology used in this research to analyze knowledge loss in other software artifacts, such as documentation
- determining the level of importance of an abandoned file.

5.2.2 Organizational volatility and post-release defects: A replication case study using data from Google Chrome

The future work that emerged from the research on the Organizational volatility and post-release defects are the following:

- A feature based analysis to understand how developer loss affects features under development
- analysing the abandoned code that was taken over by remaining developers to understand how they managed to insert fewer bugs
- comparing different granularities levels, such as directories versus files
- investigating the reason the greater the number of leaving developers the lower the number of post-release defects. We suspect the vetting of new hires plays an important role.

5.2.3 A preliminary examination of dormant files on software projects

The future work that emerged from the research on the Examination of dormant files are the following:

- Conducting a qualitative research to observe how much knowledge the developers recall about files that they have touched in the past

- Developers may modify files related to a dormant file, thereby maintaining their knowledge of the unchanged file. To understand these relationships, we would like to study file dependencies.
- investigating why files become dormant and then become active at a later date
- studying activity patterns of dormant files and creating models of dormancy.

Bibliography

[1] Christian Bird, Alex Gourley, Prem Devanbu, Anand Swaminathan, and Greta Hsu. Open borders? immigration in open source projects. In *MSR: Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 8. IEEE Computer Society, 2007.

[2] Christian Bird, Nachiappan Nagappan, Brendan Murphy, Harald Gall, and Premkumar Devanbu. Don't touch my code!: examining the effects of ownership on software quality. pages 4–14, 2011.

[3] Jesús Bisbal, Deirdre Lawless, Bing Wu, and Jane Grimson. Legacy information systems: Issues and directions. *IEEE software*, 16(5):103–111, 1999.

[4] Lionel Briand, Walcélio Melo, Carolyn Seaman, and Victor Basili. Characterizing and assessing a large-scale software maintenance organization. pages 133–143, 1995.

[5] Gerardo Canfora, Massimiliano Di Penta, Rocco Oliveto, and Sebastiano Panichella. Who is going to mentor newcomers in open source projects? page 44, 2012.

[6] M. Cataldo, A. Mockus, J.A. Roberts, and J.D. Herbsleb. Software dependencies, work dependencies, and their impact on failures. *Software Engineering, IEEE Transactions on*, 35(6):864–878, Nov 2009.

[7] Patrick A. Cataldo, Marcelo an dWagstrom, James D. Herbsleb, and Kathleen M. Carley. Identification of coordination requirements: implications for the design of collaboration and awareness tools. In *Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work*, CSCW'06, pages 353–362, NewYork,NY,USA, 2006. ACM.

[8] Marie A Cini. Group newcomers: From disruption to innovation. *Group Facilitation*, 3(2001):3–13, 2001.

[9] Stephen G Eick, Todd L Graves, Alan F Karr, James S Marron, and Audris Mockus. Does code decay? assessing the evidence from change management data. *IEEE Transactions on Software Engineering*, 27(1):1–12, 2001.

[10] H. Gall, K. Hajek, and M. Jazayeri. Detection of logical coupling based on product release history. In *Software Maintenance, 1998. Proceedings., International Conference on*, pages 190–198, Nov 1998.

[11] Xun Ge, Yifei Dong, and Kun Huang. Shared knowledge construction process in an open-source software development community: An investigation of the gallery community. In *Proceedings of the 7th international conference on Learning sciences*, pages 189–195. International Society of the Learning Sciences, 2006.

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

[13] 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.

[14] James P Guthrie. High-involvement work practices, turnover, and productivity: Evidence from new zealand. *Academy of management Journal*, 44(1):180–190, 2001.

[15] Abram Hindle, Daniel M German, and Ric Holt. What do large commits tell us?: a taxonomical study of large commits. pages 99–108, 2008.

[16] D. Izquierdo-Cortazar, G. Robles, F. Ortega, and J.M. Gonzalez-Barahona. Using software archaeology to measure knowledge loss in software projects due to developer turnover. In *System Sciences, 2009. HICSS '09. 42nd Hawaii International Conference on*, pages 1–10, 2009.

[17] A. Mockus. Succession: Measuring transfer of code and developer productivity. In *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, pages 67–77, 2009.

[18] A. Mockus, R.T. Fielding, and J. Herbsleb. A case study of open source software

development: The apache server. *ICSE: Proceedings of the 22nd international conference on Software Engineering*, pages 262–273, 2000.

[19] Audris Mockus. Organizational volatility and its effects on software defects. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, FSE '10, pages 117–126, New York, NY, USA, 2010. ACM.

[20] Audris Mockus and James D. Herbsleb. Expertise browser: a quantitative approach to identifying expertise. In *ICSE: Proceedings of the 24th International Conference on Software Engineering*, pages 503–512. ACM Press, 2002.

[21] Audris Mockus and David M. Weiss. Predicting risk of software changes. *Bell Labs Technical Journal*, 5(2), April 2000. ISSN 1089-7089.

[22] Audris Mockus and David M Weiss. Predicting risk of software changes. *Bell Labs Technical Journal*, 5(2):169–180, 2000.

[23] Audris Mockus, Randy Hackbarth, and John Palframan. Risky files: an approach to focus quality improvement effort. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 691–694, New York, NY, USA, 2013. ACM.

[24] Nachiappan Nagappan and Thomas Ball. Use of relative code churn measures to predict system defect density. In *Proceedings of the 27th International Conference on Software engineering*, ICSE '05, pages 284–292. ACM, 2005.

[25] M. Rahman and P. Rigby. Release stabilization on linux and chrome. *Software, IEEE*, pages 2–9, March-April 2015.

[26] Filippo Ricca, Alessandro Marchetto, and Marco Torchiano. On the difficulty of computing the truck factor. In *Proceedings of the 12th International Conference on Product-focused Software Process Improvement*, PROFES'11, pages 337–351, Berlin, Heidelberg, 2011. Springer-Verlag.

[27] Peter Rigby, Brendan Cleary, Frederic Painchaud, Margaret-Anne Storey, and Daniel German. Contemporary peer review in action: Lessons from open source development. *IEEE Software*, 29(6):56–61, November 2012.

[28] Peter C. Rigby and Christian Bird. Convergent contemporary software peer review practices. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 202–212, New York, NY, USA, 2013. ACM.

[29] Peter C. Rigby and Margaret-Anne Storey. Understanding broadcast based peer review on open source software projects. In *Proceeding of the 33rd international conference on Software engineering*, ICSE '11, pages 541–550, New York, NY, USA, 2011. ACM.

[30] Peter C. Rigby, Daniel M German, Laura Cowen, and Margaret-Anne Storey. Peer Review on Open Source Software Projects: Parameters, Statistical Models, and Theory. *To appear in the ACM Transactions on Software Engineering and Methodology*, page 34, August 2014.

[31] Daniel L Schacter, Daniel T Gilbert, and Daniel M Wegner. *Introducing psychology*. Macmillan, 2009.

[32] Emad Shihab, Audris Mockus, Yasutaka Kamei, Bram Adams, and Ahmed E. Hassan. High-impact defects: A study of breakage and surprise defects. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 300–310, New York, NY, USA, 2011. ACM.

[33] H. Silcock. The phenomenon of labour turnover. *Journal of the Royal Statistical Society. SeriesA(General)*, 117(4):pp.429–440, 1954.

[34] Marco Torchiano, Filippo Ricca, and Alessandro Marchetto. Is my project's truck factor low?: theoretical and empirical considerations about the truck factor threshold. In *Proceedings of the 2nd International Workshop on Emerging Trends in Software Metrics*, WETSoM '11, pages 12–18, New York, NY, USA, 2011. ACM.

[35] Quang H Vuong. Likelihood ratio tests for model selection and non-nested hypotheses. *Econometrica: Journal of the Econometric Society*, pages 307–333, 1989.

[36] L. Williams, R.R. Kessler, W. Cunningham, and R. Jeffries. Strengthening the case for pair programming. *IEEE Software*, 17(4):19–25, 2000.

[37] Robert K. Yin. *Case Study Research: Design and Methods*, volume 5 of *Applied Social Research Methods Series*. Sage Publications Inc., 3 edition, 2003.

[38] Nico Zazworka, Kai Stapel, Eric Knauss, Forrest Shull, Victor R. Basili, and Kurt Schneider. Are developers complying with the process: An xp study. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '10, pages 14:1–14:10, New York, NY, USA, 2010. ACM.

[39] Minghui Zhou and Audris Mockus. Developer fluency: Achieving true mastery in software projects. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 137–146, New York, NY, USA, 2010. ACM.

[40] Minghui Zhou and Audris Mockus. Does the initial environment impact the future of developers? In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 271–280, New York, NY, USA, 2011. ACM.

[41] T. Zimmermann, R. Premraj, and A. Zeller. Predicting defects for eclipse. In *Predictor Models in Software Engineering, 2007. PROMISE'07: ICSE Workshops 2007. International Workshop on*, pages 9–9, May 2007.