An Exploratory Study on the Influence of Developers in Technical Debt

Reem Alfayez
Center for Systems and Software Engineering
University of Southern California
alfayez@usc.edu

Pooyan Behnamghader
Center for Systems and Software Engineering
University of Southern California
pbehnamg@usc.edu

Kamonphop Srisopha
Center for Systems and Software Engineering
University of Southern California
srisopha@usc.edu

Barry Boehm
Center for Systems and Software Engineering
University of Southern California
boehm@usc.edu

ABSTRACT

Software systems are often developed by many developers who have a varying range of skills and habits. These developers have a big impact on software quality. Understanding how different developers and developer characteristics impact the quality of a software is crucial to properly deploy human resources and help managers improve quality outcomes which is essential for software systems success. Addressing this concern, we conduct a study on how different developers and developer characteristics such as developer seniority in a system, frequency of commits, and interval between commits relate to Technical Debt (TD). We performed a large-scale analysis on 19,088 commits from 38 Apache Java systems and applied multiple statistical analysis tests to evaluate our hypotheses. Our empirical evaluation suggests that developers unequally increase and decrease TD, a developer seniority in a software system and frequency of commits are negatively correlated with the TD the developer induces, and a developer commit interval has a positive correlation with the TD the developer induces.

CCS CONCEPTS

• **Social and professional topics** → **Software maintenance**; • **Software and its engineering** → **Maintaining software**;

KEYWORDS

Software Engineering, Software Maintenance, Technical Debt, Project Management,Human Factors, Developer Experience, Developer Contribution

ACM Reference Format:
Reem Alfayez, Pooyan Behnamghader, Kamonphop Srisopha, and Barry Boehm. 2018. An Exploratory Study on the Influence of Developers in Technical Debt. In *TechDebt '18: TechDebt '18: International Conference on Technical Debt , May 27–28, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3194164.3194165

TechDebt '18, May 27–28, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5713-5/18/05…$15.00
https://doi.org/10.1145/3194164.3194165

1 INTRODUCTION

The developers of a software system play an important part in its success. Studies have shown that human factors such as organizational structure, code ownership, experience, and emphasis on producing more functionalities have a huge impact on software quality [5, 12, 19, 20], and that issues degrading software quality have a negative impact on its maintainability and its end users satisfaction with its quality and the services that it delivers [5, 17].

The inescapable increase in software demand, complexity, and the pace of changes in software technology has introduced challenges in managing software quality, preventing software defects, and producing maintainable code. Software defects costed the worldwide $1.1 trillion and affected 4.4 billion people in 2016 [1], on average each Java LOC has $5.42 of TD [11], and approximately 75% of most systems' lifecycle costs is spent on software maintenance.

This places great emphasis on analyzing and addressing the root causes of expensive software maintenance, defective software, and poor software quality. Being able to identify which developers and developer characteristics enhance and reduce the software maintenance and quality would be an invaluable resource for developers and software mangers in distributing tasks, conducting peer reviews, focusing quality assurance efforts, and manging quality budgets effectively.

Providentially, the advances in technology produce many tools that can increasingly provide ways to detect maintainability and software quality issues and track their sources to understand what factors might relate to the occurrence of these issues. Version Control Systems (VCS), static code analysis, and data analytics help software managers and developers assess the buildup of their software maintainability issues, such as Technical Debt (TD).

To address these needs, we leveraged the rich and unique information that VCS provides in each commit such as commit time and author, and we utilized static code analysis to assess software quality before and after each commit to preform a commit-impact analysis [3]. This reveals how each change impacts software quality and helps exploring how different developers and different developer characteristics relate to TD as a measurement of the software quality and maintainability. We performed a large-scale analysis on 19,088 commits from 38 Apache Java systems from multiple software domains to calculate the amount of TD after each system change which enabled us to track the impact of each developer on

TD. We applied multiple statistical analysis tests to evaluate our hypotheses. The main contribution of this study is finding that:

(1) Developers are unequally responsible for increasing and decreasing TD.
(2) Developer seniority in a software system has a negative association with added TD.
(3) Developer commit frequency has a negative association with added TD.
(4) Developer interval between commits has a positive association with added TD.

The remainder of the paper is organized as follows: Section 2 presents the background for this study; Section 3 describes the setup for our empirical study; Section 4 presents our analysis and results; Section 5 discusses the implications of our study; Section 6 presents the threats to validity; Section 7 highlights some related work. Finally, we conclude and present the future work in Section 8.

2 BACKGROUND

In this section, we present background information that helps in understanding the context of our study: open source software, version control systems, and static code analysis.

2.1 Open Source Software Systems

2.1.1 Definition. The Free Software Foundation defines free software as software that grants the users and community the freedom to run, copy, distribute, study, change, and improve the software [31]. It is sometimes referred to as Free and Open Source Software (FOSS) [27] . Many thousands of Open Source Software (OSS) systems exist today and they are becoming more popular in use. SourceForge[1], an open source community resource, has more than 4,800,000 downloads a week. It hosts over 430,000 OSS systems that fall under different domains and are written using multiple programing languages. The Apache Software Foundation[2] has 306 software systems that also fall under multiple domains and programming languages. OSS systems are also popular in use; the Apache server is used in 49.5% of all the websites whose webserver is known [30] and Unix is used by 66.5% of all the websites whose operating system is known [29].

2.1.2 Characteristics and Development Process. OSS has a unique development process; it is developed and maintained by volunteer contributors who are geographically distributed and have the freedom to pick tasks they like to work on. In most OSS systems, the contributors of a system are also its users, and there is no central management that controls all the developers activities [32]. Contributors coordinate their work through versioning systems, issues tracking systems, and the project mailing list [24]. These tools allow collaborative working among software developers since they enable easy code sharing, and they keep track of each change performed on a software during its development and maintenance.

2.1.3 Developers. The OSS systems are usually started by a company open sourcing a software system, a single developer or a group of developers. As the new software system becomes popular and used, many users and developers want to contribute to it for different motives [32]. Some contributors contribute to improve the software system itself by fixing a bug or adding support for their native language. Others see the contribution as a means to improve themselves by learning from doing or gaining professional opportunities from their contributions [4]. As the software system grows, it will have an active community that is usually highly diverse in terms of competence, skill, experience involvement, and engagement levels.

[1] https://sourceforge.net/
[2] http://www.apache.org/

2.2 Version Control Systems

2.2.1 Definition. VCS, such as Git, Concurrent Versioning System (CVS), and Subversion (SVN) records changes to a file or set of files over time; they record the change, the time of the change, and who performed the change. This allows its users to revert selected files or a whole project back to a previous state and compare changes over time [8, 13, 26]. It facilitates collaborative software development processes since it allows developers to distributedly and simultaneously work on the same software system [13]. While VCS can be used in any domain and different areas of work, we are only interested in VCS in collaborative software systems development context. As a result, we will be referring to the user as a developer in this section.

2.2.2 Components.

Repository. The repository is the core of the VCS; it stores the system's data in the form of the file system tree. Developers connect to the repository to write or read. When developers write to these files they make their changes available for others, and when they read, they get the changes of others. The repository keeps track of each version of these files and who made these changes and when; this allow developers to access any version at any time or revert the file to one of its previous versions [9].

The Working Copy. A working copy is a local copy of a particular version of the developed software system upon which a developer can work in and change. It appears to developers in their machines as any other local directory full of files, and the developers can write to and read these files as they will do with any other regular local files. Managing the working copy and transferring changes made to it to and from the repository is handled by the VCS client software [9].

Branches. A branch is a line of development that exists independently of the main development line, yet it shares the common history of the main line. The new created branch inherits the history of the development line that it originated from, then it starts to create its own history [9]. Figure 1 shows that a copy of the main development line was created to fix a bug; the new created branch inherited the history of the main development line and it started to generate its own history. When the development was completed in the bug fix branch, it was merged in with the main development line. Another branch was also created to work on a new feature, and it was not merged back to the main development line. Branching is one of the core ideas in VCS; it allows developers to work on different parts of the system in parallel independently without effecting the main development line, then merging their changes to the main development line [8].

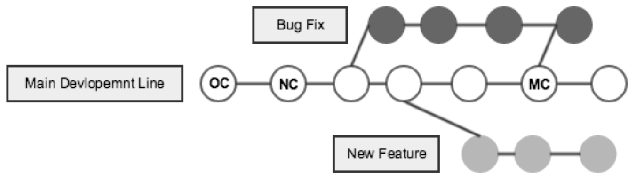

Figure 1: Branching in version control systems.

Commits. In VCS, developers write their changes on the project to the repository through commits. Each commit records what changed, the time of the commit, the developer basic information, and a commit message. This information helps in understanding how a software system evolves overtime.

2.3 Static Code Analysis

Static code analysis is the process of analyzing a software without executing it [28]. It can be performed on a source code which is the exact reflection of the software itself or on a compiled bytecode [16]. The analysis reveals software defects that are not visible to the compiler such as bugs, coding guidelines violations, or design issues [28]. It can be considered as machine-assisted code review that helps in detecting developers mistakes at early stages. SonarQube is an open source quality management platform that helps developers continuously improve their source code. It implements the SQALE method and measures code quality based on seven axes of quality: design/architecture, duplications, comments, unit tests, complexity, potential bugs, and coding rules [7, 15]. It has its predefined set of rules and users can extend these coding rules by adding their own custom ones [7]. While SonarQube provides a wide set of code analyzers that supports over twenty programming languages including Java, C, C++, C#, PHP, and JavaScript, not all of these analyzers are released as free software. Each analyzer provides numerous rules to detect general and language-specific quality issues. Developers can integrate these analyzers into different development environments or run the SonarQube analyzer, a Java-based command-line tool, to analyze their code to ensure continuous inspection through the development process. This will make code quality analysis and reporting an integral part of the development lifecycle [7]. SonarQube reports the results in a dashboard as Figure 2 shows. The dashboard can be customized and edited by the developers. In this study, we measured TD principle using SonarQube. It measures TD principle based on several aspects of code quality rules ans standards. Each SonarQube rule that detects a maintainability issue has a remediation effort function. This remediation effort is used to compute the total TD in a system by simply summing the remediation effort of every maintainability issue.

3 EMPIRICAL STUDY SETUP

Our study aims to explore how different developers and developer characteristics influence the software system with regards to its quality and maintainability using TD as a measurement. The next two subsections describe our research questions and our data collection process.

3.1 Research Questions

RQ1: Is TD uniformly changed by all developers within an OSS system?

- **RQ1.1:** Is TD uniformly increased by all developers within an OSS system?
- **RQ1.2:** Is TD uniformly decreased by all developers within an OSS system?

Since software system developers have highly diverse ranges of competence, skill, experience, involvement, and engagement levels, they might have different impact on the software system TD. We hypothesize that developers unequally add or remove TD in a software system.

RQ2: How does a developer commit frequency in a system relate to the amount of TD that the developer introduces per commit?

Software system developers have different ranges of involvement, contribution, and engagement within a system. One widely used approximation of developers involvement and contribution is the frequency of their commits (i.e, developers who commit more contribute more, and are more involved within the system). Having a high involvement within a system implies that a developer is more familiar and experienced with it [6, 12, 22, 23]. In other words, a developer who frequently interacts with the system has a better understanding of its structure and architecture, and learns the best practices to fix and avoid defects. Thus it is likely that the changes that this developer makes would be of a higher quality and add less TD. Our hypothesis is that the more a developer commits, the less TD the developer adds per commit.

RQ3: How does a developer seniority in a system relate to the amount of TD that the developer introduces per commit? During the software system's life cycle, many developers join and leave a project. Oftentimes, senior developers (i.e., the longer a developer is in a project, the more senior the developer is) have more interaction and experience within a project; they know the technology, strategy, the plan of work, and goals hence their contributions are usually associated with superior quality [6, 12, 22, 23]. We hypothesize that the longer a developer contributes to a software system, the less TD the developer adds per commit.

RQ4: How does a developer commit interval in a system relate to the amount of TD that the developer introduces per commit?

Although developers learn and accumulate experience about the system over time [6, 12, 22, 23], the knowledge usually depreciates when the task is not performed for long; Humans tend to forget information over time when they do not try to retain it [33]. In software development, developers might be contributing to a project, stop working on it for a period then return to work. One measurement of the time between developers contribution is the interval between a developer commits. Our hypothesis is that the longer a developer commit interval is, the more TD the developer adds per commit.

3.2 Data Collection

Subject Systems: OSS systems were selected from the Apache Software Foundation. The selection and analyzing process was fully

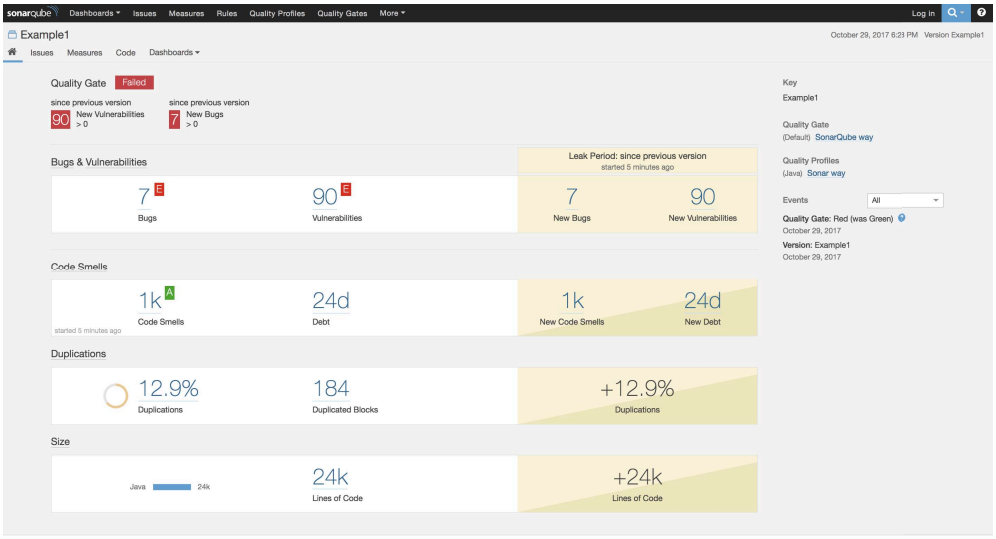

Figure 2: SonarQube dashboard.

automated by utilizing GitHub API[3] and Bash scripts to ensure employing consistent criteria for selecting our subject software systems to confirm the quality of our study. We fetched the name, the number of commits, the repository URL, the programming language, and the latest update date of all Apache software systems using GitHub API. Then we only considered OSS systems that satisfies all of the following criteria to ensure relatively a uniform sample:

- There is more than one official release.
- The programming language is Java.
- The Git repository is active, and has an update in 2017.
- The software system has less than 3,000 commits.
- The system has one module that contains most of the source code.

Table 1 summarizes the characteristics of the selected systems. For each considered system, we obtained its commits source code and their information such as date, author, and previous commit via the GitHub API, then we analyzed the source code resulted from each commit using SonarQube to calculate its TD.

Measuring Change: to calculate change in TD between each commit and its parent commit (i.e., a commit immediate ancestor in the development history), we categorized commits into three types based on their number of parents:

- Normal: a commit that has only one parent. "NC" in Figure 1
- Merge: a commit that has two or more parents. "MC" in Figure1
- Orphan: an initial commit that has no parents. "OC" in Figure 1

To measure change produced by a normal commit, we measured TD using Sonarqube in the normal commit and its parent. Then we calculated the difference between the measured values to quantify change in TD resulted from that specific commit. Merge commits integrate different branches into a single destination branch. They do not produce change; they just transfer change previously committed to the merged branches into one destination branch. Since we are interested in studying change introduced by one developer in a single commit, we only consider them as parent commits in our study. The orphan commit is the initial version of the project which might not contain the source code. If the orphan commit contains the source code, we measured its TD to use its value in measuring TD change in its descendant commit. If the orphan commit does not contain the source code, we did not include it in our analysis; a descendant normal commit will contain the initial source code, and we considered it as an orphan commit.

Variables: We summarize and explain multiple variables used throughout this study:

• ***Developer Increased TD:*** in each system, we obtained the commits that increased TD. Then for each developer who increased TD at least in one commit, we calculated the Developer Increased TD by summing the total amount of TD introduced in each commit the developer authored then normalized that by dividing the obtained summation by total number of LOC change using the following formula:

$$Developer\_increased\_TD\,(d_i) = \frac{\sum(TD\_increase(d_i))}{\sum(LOC\_change(d_i))}$$

Where:

d_i : a developer i who increased TD in a commit.

$TD\_increase\,(d_i)$: total amount of TD increased by developer d_i in a system.

$LOC\_change\,(d_i)$: total amount of LOC changed by the developer d_i in a system.

We believe that obtaining the normalized value gives a better insight and fair assessment since the amount of TD that is increased by a developer can be highly influenced by the LOC a developer contributes.

[3]https://developer.github.com/v3/

Table 1: Characteristics of subject OSS systems

System	Domain	Time span	Million lines of code (LOC)
Avro	Data Serialization	07/11-01/17	2.43
Calcite	Data Management	07/14-02/17	75.46
C-BCEL	Bytecode Eng.	06/06-12/16	16.5
C-Beanutils	Reflection Wrapper	07/07-11/16	1.71
C-Codec	Encoder/Decoder	09/11-09/16	2.32
C-Collections	Collections Ext.	03/12-10/16	13.87
C-Compress	Compress Lib.	07/08-02/17	16.84
C-Configuration	Conf. Interface	04/14-01/17	8.83
C-CSV	CSV Library	11/11-02/17	0.68
C-Dbcp	DB Conn. Pooling	12/13-11/16	2.06
C-IO	IO Functionality	01/02-12/16	5.99
C-JCS	Java Caching	04/14-02/17	3.66
C-Jexl	Expression Lang.	08/09-10/16	2.26
C-Net	Clientside Protocol	08/06-02/17	14.96
C-Pool	Object Pooling	04/12-02/17	1.27
C-SCXML	State Chart XML	03/06-08/16	2.53
C-Validator	Data Verification	07/07-02/17	1.63
C-Vfs	Virtual File System	11/06-01/17	13.32
CXF-Fediz	Web Security	04/12-03/17	0.89
Drill	SQL Query Engine	04/15-02/17	58
Flume	Data Collection	08/11-11/16	2.14
Giraph	Graph Processing	11/12-01/17	15.12
Hama	BSP Computing	06/08-04/16	7.77
Helix	Cluster MNGMT	01/12-06/16	27.36
HTTPClient	Client-side HTTP	03/09-01/17	16.54
HTTPCore	HTTP Transport	03/09-02/17	7.38
Mina	Socket Library	11/09-04/15	2.1
Mina-SSHD	SSH Protocols	04/09-02/17	24.66
Nutch	Web Crawler	03/05-01/17	17.85
OpenNLP	NLP Toolkit	04/13-02/17	18.85
Parquet-MR	Storage Format	02/13-01/17	5.2
Phoenix	OLTP Analytics	01/14-02/17	131.13
Qpid-JMS	Message Service	02/15-02/17	9.12
Ranger	Data Security	03/15-02/17	19.97
Santuario	XML Security	01/11-02/17	16.38
Shiro	Java Security	03/09-11/16	3.26
Tiles	Web Templating	07/06-07/16	1.71
Zookeeper	Distributed Comp.	06/08-02/17	14.98

• ***Developer Decreased TD:*** in each system, we obtained commits that decreased TD. Then for each developer who decreased TD at least in one commit, we calculated the Developer Decreased TD by summing the total amount of TD removed in each commit the developer authored then normalized that by dividing the obtained summation by total number of LOC change using the following formula:

$$\boldsymbol{Developer\_decreased\_TD\,(d_i)} = \frac{\sum(TD\_decrease(d_i))}{\sum(LOC\_change(d_i))}$$

Where:

$\boldsymbol{d_i}$: a developer i who decreased TD in a commit.

$\boldsymbol{TD\_decrease\,(d_i)}$: total amount of TD decreased by developer d_i in a system.

$\boldsymbol{LOC\_change\,(d_i)}$: total amount of LOC changed by developer d_i in a system.

• ***Induced TD:*** to calculate induced TD for each commit, we simply measured the change in the amount of TD by subtracting its TD amount from its ancestor TD amount as measured by SonarQube. Then normalized that by dividing the obtained TD change by total number of LOC change using the following formula:

$$\boldsymbol{Induced\_TD\,(c_i)} = \frac{TD(c_i) - TD(c_{i-1})}{LOC\_change(c_i)}$$

Where:

$\boldsymbol{TD\,(c_i)}$: amount of TD in the current commit.

$\boldsymbol{TD\,(c_{i-1})}$: amount of TD in the current commit's ancestor.

$\boldsymbol{LOC\_change\,(c_i)}$: amount of LOC changed by the current commit.

• ***Developer Commits Frequency:*** to measure a developer commits frequency in a system, for each commit we summed its author previous commits up to but not including the current commit using the following formula:

$$\boldsymbol{Frequency\,(c_j, d_i)} = \sum_{k=1}^{num\_of\_commits(c_j, d_i)} 1$$

Where:

$\boldsymbol{c_j}$: current commit j.

$\boldsymbol{d_i}$: a developer i who authors a commit c_j.

$\boldsymbol{num\_of\_commits\,(c_j, d_i)}$: total number of previous commits authored by d_i in the system up to but not including c_j .

• ***Developer Seniority:*** to measure a system developer seniority in days, for each commit, we calculated its author seniority by subtracting the commit date from its author first commit date in the system using the following formula:

$$\boldsymbol{Seniority\,(c_j, d_i)} = Commit\_date\,(c_j) - First\_commit\_date\,(d_i)$$

Where:

$\boldsymbol{c_j}$: a commit j which we want to measure its author seniority.

$\boldsymbol{d_i}$: a developer i who authors a commit c_j.

$\boldsymbol{Commit\_date\,(c_j)}$: the date of the commit c_j.

$\boldsymbol{First\_commit\_date\,(d_i)}$: the date of the first commit authored in the system by developer d_i.

• ***Developer Commits Interval:*** to measure a developer commits interval in a system, for each commit, we calculated its author commits interval in minutes by subtracting the current commit time from the developer previous commit time in the system using the following formula:

$$\boldsymbol{Interval\,(c_j, d_i)} = Commit\_time\,(c_j) - Previous\_commit\_time\,(d_i)$$

Where:

$\boldsymbol{c_j}$: a commit j which we want to measure its author commit interval.

$\boldsymbol{d_i}$: a developer i who authors a commit c_j.

$\boldsymbol{Commit\_time\,(c_j)}$: the time of the commit.

$\boldsymbol{Previous\_commit\_time\,(d_i)}$: the time of the previous commit authored in the system by developer d_i.

4 DATA ANALYSIS AND RESULTS

To address our research questions, we applied multiple statistical analysis techniques. In this section, we summarize the techniques we used and our results.

To answer RQ1, for each system, we obtained Developer Increased TD and Developer Decreased TD as explained in Section 3. Then we compared the distributions of these two variables among system developers using the Gini index following the methodology explained in [2]. The Gini index or the Gini coefficient is a statistical measure of distribution that is commonly used in economic to measure the income distribution. The coefficient ranges between 0 and 1, where 0 represents complete equality and 1 represents complete inequality [10]. In the context of **RQ1.1**, an Increased TD Gini index equals to zero implies that all developers increased the

same amount of TD. On the contrary, an Increased TD Gini index equals to one, implies that only one developer increased TD more than any other developer in the system.

Table 2 summarizes the number of developers considered in each system and the system Increased TD Gini Index. While all of our subject systems have an unequal distribution of the Developer Increased TD among their developers as Table 2 shows, some of the systems such as C-Net and Santuario have a very low Increased TD Gini Index (≤ 0.1) which implies that developers in these two systems almost equally produce the same amount of normalized TD. On the other hand, Parquet-MR and C-Validator have a high Increased TD Gini Index (≥ 0.80) which implies that some developers in these two systems introduce more normalized TD than other system developers. As Figure 3 shows, developer 4 and developer 9 produced around 50 times and 10 times more normalized TD than other developers in C-Validator system respectively. In the

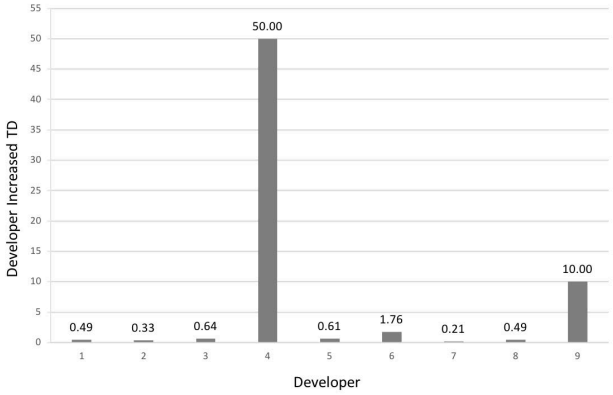

Figure 3: A barchart for the distribution of Developer Increased TD among developers in C-Validator.

context of **RQ1.2**, a Decreased TD Gini index equals to zero implies that all developers have the same amount of Decreased TD. On the contrary, a Decreased TD Gini index equals to one implies that only one developer is responsible for decreasing TD in the system. Table 2 summarizes the number of developers considered in each system and the system Decreased TD Gini index.

As Table 2 shows, the Decreased TD Gini Index for C-Configuration, C-Dbcp, and Qpid-JMS is very small (≤ 0.1) which implies that developers in these systems almost equally decreased the same amount of normalized TD. On the other hand, Calcite, C-Compress, Mina-SSHD, OpenNLP, and Phoenix have a high Decreased TD Gini index (≥ 0.80) which implies that some developers in these systems decreased more normalized TD than other system developers.

To answer RQ2, for each commit, we calculated its Developer Commits Frequency and Induced TD as described in Section 3. Figure 4 shows that most of the developers do not commit often since the graph is more skewed to the left. Then to assess the relation between the Developer Commits Frequency and Induced TD, we applied Spearman's correlation test since it is less sensitive to outliers which exist legitimately in our data (i.e., being part from the observation) [18]. Spearman's correlation test shows a

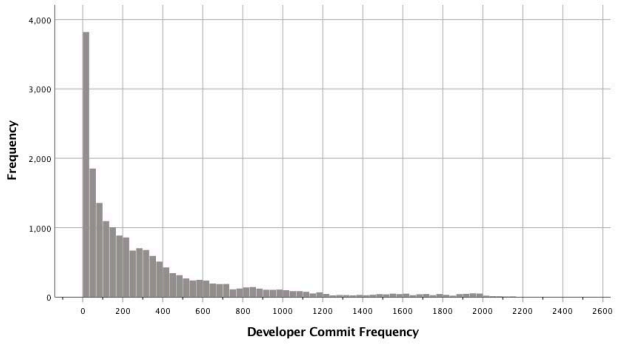

Figure 4: A histogram for the frequency distribution of developers commit frequency.

significant low negative correlation between Induced TD and the Developer Commits Frequency ($r = -0.026, p = 0.000302$) which implies that developers who commit more frequently tend to add less TD compared to developers who commit less to the system.

To evaluate RQ3, for each commit, we calculated its Developer Seniority and Induced TD as described in Section 3. Figure 5 shows that the number of commits decrease as the seniority of a developer increases which might implies that not many developers continue contributing to a system. Then we performed Spearman's corre-

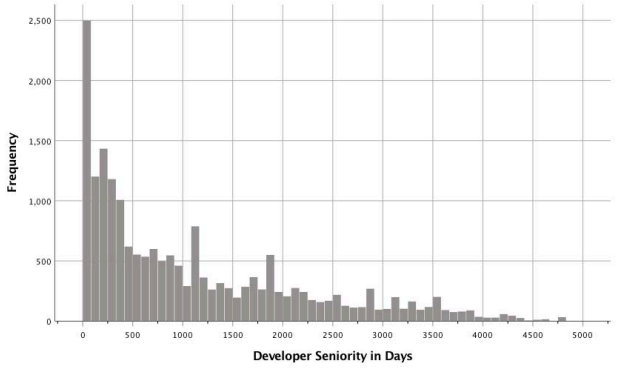

Figure 5: A histogram for the frequency distribution of developers seniority in days.

lation test to assess the relation between Developer Seniority and Induced TD. Spearman's correlation test shows a significant low negative correlation between Developer Seniority and Induced TD ($r = -0.07, p = 2.1375E-22$) which denotes that the more senior a developer becomes in a system, the less TD the developer tends to accumulates.

To answer RQ4, for each commit, we calculated its Developer Commits Interval and Induced TD as described in Section 3. Figure 6 shows that developers tend to have shorter interval periods between their commits. In other words, as the interval period increases, the number of developer commits decreases and vice versa.

Table 2: Increased TD Gini index, Decreased TD Gini index, and the number of considered developers in each subject system

Application	Increased TD Gini index	Increase # of developers	Decreased TD Gini index	Decrease # of developers
Avro	0.46	14	0.36	4
Calcite	0.47	48	0.84	22
C-BCEL	0.3	7	0.39	7
C-Beanutils	0.68	5	0.33	4
C-Codec	0.37	5	0.47	4
C-Collections	0.44	9	0.41	5
C-Compress	0.29	18	0.83	12
C-Configuration	0.44	2	0	2
C-CSV	0.27	5	0.2	5
C-Dbcp	0.26	4	0.08	5
C-IO	0.44	18	0.69	14
C-JCS	0.51	3	0.29	4
C-Jexl	0.42	2	0.4	3
C-Net	0.08	4	0.3	5
C-Pool	0.70	4	0.62	4
C-SCXML	0.28	6	0.31	6
C-Validator	0.80	9	0.33	7
C-Vfs	0.43	10	0.75	7
CXF-Fediz	0.34	3	0.16	3
Drill	0.47	47	0.73	29
Flume	0.52	19	0.5	11
Giraph	0.45	19	0.54	17
Hama	0.48	11	0.4	7
Helix	0.54	18	0.55	10
HTTPClient	0.42	9	0.57	7
HTTPCore	0.42	4	0.47	4
Mina	0.66	4	0.17	5
Mina-SSHD	0.47	11	0.82	7
Nutch	0.43	24	0.51	18
OpenNLP	0.48	14	0.9	11
Parquet-MR	0.80	24	0.69	16
Phoenix	0.62	48	.87	29
Qpid-JMS	0.32	3	0.03	2
Ranger	0.3	18	0.48	18
Santuario	0.09	3	0.1	3
Shiro	0.4	5	0.16	3
Tiles	0.34	4	0.79	5
Zookeeper	0.48	23	0.78	12

Then we performed Spearman's correlation test to assess the relation between the Developer Commits Interval and Induced TD. Spearman's correlation test shows a significant positive correlation between the Developer Commits Interval and Induced TD ($r = 0.10, p = 1.562E-41$). This indicates that the longer the interval between a developer consecutive commits, the more TD the developer adds to the system.

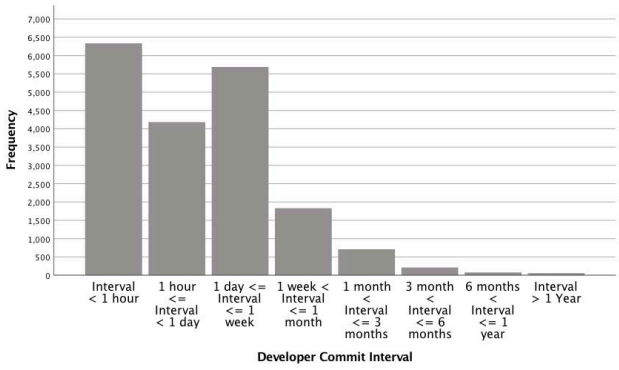

Figure 6: A histogram for the frequency distribution of developers commit interval.

5 IMPLICATIONS

The analyses presented in our study for OSS systems can be applied by any organization wishing to improve its software and its software engineering by having its projects gather and analyze the types of data presented here. Researchers also can find it beneficial in multiple ways. In this section, we summarize the implications of our study.

At the management level, managers can determine which of their project types have better or worse quality, and how the quality correlates with customer satisfaction and with the total cost of ownership. Additionally, managers can better understand which types of personnel contribute most to accumulating or removing TD. Additionally, gaining such an insight can be constructive for resource allocation; managers can refer to developers performance in pursuance of proper human resources deployment to improve quality outcomes, assigning developers training to amplify skills, and distributing quality assurance efforts and budget.

At the personal level, developers can get a better insight into how their changes impact the quality of the system, and how they perform compared to other system developers which might be beneficial for self assessment and improvement. Moreover, since the analysis helps in identifying developers who improve the system quality, other developers might seek help and advice for software quality improvement tips from them. Further, developers might be more meticulous, seek peer reviews, or spend more time understanding the system after not contributing for a long period or when newly contributing to a system given our findings.

At the research level, the results of our study indicate that some developers contribute in improving the quality of the system and other developers harm the software quality more than other developers. Applying an individual assessment on these developers might reveal more information about the characteristics of developers who increase or decrease TD which can be used to build a developers profile that may aid in managing TD, assign code review tasks, and manage pair programming activities. Additionally, our results show that the newer a developer is, the less commit frequency a developer has, or the longer the interval between commits a developer authors, the more TD the developer produces. This can be used in designing and implementing tools that can identify quality risk commits (i.e., commits authored under conditions promoting the introduction of TD) that can be the focus of quality assurance activities.

6 THREATS TO VALIDITY

This section identifies several potential threats to the validity of our study with their corresponding mitigations based on the classification by [25].

Construct Validity: Our study aims to explore whether TD is introduced and decreased uniformly among the system developers and how the developers commit frequency, seniority , and interval between commits in a software system relate to the TD the developers introduce. A threat to construct validity in our study involves the accuracy of measuring TD. To mitigate this threat, we chose SonarQube which is a widely utilized tool to calculate TD, and to the best of our knowledge, it is the only open source tool that identifies and evaluates TD [7, 15].

Internal Validity: Threat to internal validity relates to whether the results in our study truly follow from our data. Particularly, whether the metrics are meaningful to our conclusions and whether the measurements are adequate. To avoid bias and error in our study, we avoided using manual methods and human-judgment in assessing the amount of TD. We relied on SonarQube to measure TD instead of other methods such as self-admitted technical debt (SATD) [21]. This ensures that all our subject systems were assessed using one unified set of rules. We used the default rule set that SonarQube provides to avoid bias in weighing the importance of one rule over another or to introduce coding rules and standards that are not followed by the community.

External Validity: Threat to external validity concerns the generalizability of our results. The results and conclusions of our empirical study were based on a set of commits of 38 Apache Java software systems. The size of our selected data set is smaller than the population of all OSS systems which could impact the generalizability of our conclusions. To mitigate this threat, we ensured that all our subject systems are well-known systems in one of the leading OSS communities. Additionally, these systems are continuously evolving and vary among various dimensions, such as domain, size, timeframe, and number of versions. However, we acknowledge several limitations to the generalizability of our conclusions. First, our study involved only Apache Java systems, so the results may not generalize to other programming languages. However, we expect that since the contribution and development patterns are similar, we would arrive at similar conclusions. Second, our systems selection was biased towards well-known and widely-used systems. It is possible that less used and unpopular systems may have different patterns. We hypothesize that since these systems are less successful, it is likely that they have higher TD, lower software quality, and fewer developers contribution. Exploring these hypotheses is something we intend to do in the future to investigate how variations in systems usage and popularity relate to developers degree of involvement and quality of contribution. Lastly, our study only included OSS systems. We acknowledge that closed source systems that satisfies our selection criteria might have different patterns and results. Nonetheless, we were unable to find sources of data outside of the OSS community to test our hypotheses; even though some companies are performing such studies, mostly the results are confidential.

Reliability: The reliability of a study is measured by the reliability and reproducibility of its findings. We strove to eliminate any threats to this aspect of validity by reporting the data collection method, subject systems, statistical tests, and results. We believe that any other researchers will obtain the same results as long as they follow the same methodology and use the same data set.

7 RELATED WORK

This section presents research efforts focusing on exploring how different developers characteristics relate to the quality of their contribution. We discuss the similarities and differences of our work with the related work.

Eyolfson et al. [12] studied Linux kernel, PostgreSQL, and the Xorg server to investigate the correlation between commit bugginess and the time of day the commit authored, the day of week of the commit authored, the commit author seniority in a project measured in days as the interval between the author first commit and the time of the current commit, the author commit frequency, and whether or not the commit was marked as stable. They found that late-night commits are significantly buggier, commits authored between 8 a.m and noon are less buggy, the bugginess of commits versus day-of-week varies among systems, code bugginess decreases with increased author experience, daily committing developers tend to produce less-buggy code, and stable commits are significantly less buggy than commits in general. We also investigated the correlation between code quality and its author seniority and commit frequency in a system. However, we measured the code quality using TD while the researchers in [12] measured the code quality by identifying bug-fixing commits and their lines change

then tracing the previous commits to find the commits that introduced these bugs. Additionally, we measured commit frequency of each developer as the number of developer previous commits up to the current commit while [12] classified developers according to commit frequency (i.e., most-common interval between consecutive commits daily, weekly, monthly, other, or single).

Qiu et al. [22] measured developer quality as the percentage of the developer commits that do not introduce bugs to the developer total commits. Then they examined how developer quality is distributed, evolves with software evolution, correlates with developer contribution measured as modified lines of code, and correlates with developer project ownership measured as the ratio of number of commits that the developer authores by analyzing the commits of latest released version of 6 OSS systems. The results show that developers with different quality levels are evenly distributed and developer quality usually increases slightly as the software evolves. Nonetheless, there is no clear relation between developer contribution and ownership and developer quality since some of the correlations are not significant. Our work also studied how developer quality is distributed in a system. However, we relied on the amount of increased and decreased TD to measure the developer quality, and the Gini index to measure the distribution rather than the complementary cumulative distribution function. We also explored how a developer seniority in a system relates to the developer quality. However, we investigated that by using Spearman's rank correlation as opposed to plotting a line chart and observing the trend.

Amanatidis et al. [2] investigated the distribution of induced TD among OSS system developers, the types of violations introduced by each developer, and the relation between a developer maturity and the amount of TD per line the developer induces by analyzing four widely-used PHP OSS systems. Their results showed that the distribution of induced TD is highly imbalanced among the system developers, and different developers introduce different types of TD. Additionally, they found no significant correlation between developers maturity measured as time between a developer first commit and last commit in the OSS system and the amount of TD per line the developer induces. Similarly, we investigated the distribution of TD among OSS system developers. However, our study focus is Java OSS systems, and our sample size is much larger (i.e., 38 systems). Additionally, the researchers investigated the distribution of induced TD which can be positive or negative. Our study investigated the distribution of increased TD and decreased TD separately which can reveal which developers undermine the quality of the system and which developers improve the quality of the system. We also explored how the developer seniority in a system relates to the amount of normalized induced TD. However, we measured the developer seniority as the difference between the developer current commit date and the developer first commit date in the system. Then we tested the correlation of a developer seniority with the amount of TD induced by the current commit.

Rahman and Devanbu [23] studied 4 OSS systems to understand the impact of developer file ownership, developer specialized experience in a file (i.e., lines of deltas contributed to a file) , and general experience in a system (i.e., lines of deltas contributed to the system) on software defects. They found that defected code is less likely to involve contributions from multiple developers, file owners write significantly less defected code in their files, lower developer specialized experience is associated with higher defected code, and developer general experience has no clear association with defected code. Our work also investigated the relation between developer experience and the developer code quality. However, we measured developer experience as the commit frequency and developer seniority within a system, and we relied on TD as a quality measurement.

Alves et al. [14] studied how different types of developers relate to the introduction of code smells by analyzing 6000 commits of 5 open source object-oriented systems. They categorized developers into 5 groups based on the frequency of their contributions and authorship measured by the amount of number of modified files and lines of code. Their results show that developers in groups that participate less introduce and remove more code smells than developers who belong to more frequent participation groups. Similarly, we explored how a developer commit frequency relates to the developer contribution quality. However, we rely on the amount of induced TD instead of code smells as a proxy of the contribution quality, and our investigation was based on each single developer as opposed to developers in groups.

8 CONCLUSION

Software systems are the artifacts of multiple developers collaborations. These developers have varying levels of experience and involvement within a software system. Having automated tools that measure the quality of the produced software enables measuring the quality of change produced by each developer which provides a wealth of information that aids in tracking what type of change and developers affect the quality of a software system.

In this paper, we investigate whether TD is increased and decreased by all system developers equally and how the developer commit frequency, seniority, and interval between commits in a software system relate to the TD the developer introduces by analyzing a total of 19,088 commits from 38 Apache Java software systems. Our results show that in all of our subject systems some developers are responsible for increasing TD more than others. Similarly, in the vast majority of our subject systems (over 97%) some developers decreased TD more than other developers. Additionally, the study reveals that a developer commit frequency and seniority in a software system have a significant negative relationship with the TD introduced by the developer. However, a developer interval between commits has a significant positive relationship with the TD that the developer produces.

In the future, we will further our study to understand the characteristics of developers and commits that increase and decrease TD, and expand our investigation by including a larger number of OSS systems as well as closed source software systems, more programming languages, and more domains. Our ultimate goal is to identify what type of developers and changes increase or decrease TD which would be an invaluable resource for decision makers and developers to help them study the trade space by providing what factors introduce and decrease TD to the software system.

ACKNOWLEDGMENTS

This material is based upon work supported in part by the U.S. Department of Defense through the Systems Engineering Research Center (SERC) under Contract H98230-08-D-0171. SERC is a federally funded University Affiliated Research Center managed by Stevens Institute of Technology.

REFERENCES

[1] 2016. Software Fail Watch: 2016. (2016). Retrieved January 19, 2018 from https://www.tricentis.com/resource-assets/software-fail-watch-2016/
[2] Theodoros Amanatidis, Alexander Chatzigeorgiou, Apostolos Ampatzoglou, and Ioannis Stamelos. 2017. Who is Producing More Technical Debt? A Personalized Assessment of TD Principal. (2017).
[3] P. Behnamghader, R. Alfayez, K. Srisopha, and B. Boehm. 2017. Towards Better Understanding of Software Quality Evolution through Commit-Impact Analysis. In *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. 251–262. https://doi.org/10.1109/QRS.2017.36
[4] Hind Benbya and Nassim Belbaly. 2010. Understanding developers' motives in open source projects: a multi-theoretical framework. *Communications of the AIS* 27, 30 (2010), 589–610.
[5] Barry W Boehm, John R Brown, and Myron Lipow. 1976. Quantitative evaluation of software quality. In *Proceedings of the 2nd international conference on Software engineering*. IEEE Computer Society Press, 592–605.
[6] Frederick P Brooks Jr. 1995. The mythical man-month (anniversary ed.). (1995).
[7] G. Ann Campbell and Patroklos P. Papapetrou. 2013. *SonarQube in Action* (1st ed.). Manning Publications Co., Greenwich, CT, USA.
[8] Scott Chacon and Ben Straub. 2014. *Pro git*. Apress.
[9] Ben Collins-Sussman, Brian Fitzpatrick, and Michael Pilato. 2004. *Version control with subversion*. " O'Reilly Media, Inc.".
[10] Frank Cowell. 2011. *Measuring inequality*. Oxford University Press.
[11] Bill Curtis, Jay Sappidi, and Alexandra Szynkarski. 2012. Estimating the principal of an application's technical debt. *IEEE software* 29, 6 (2012), 34–42.
[12] Jon Eyolfson, Lin Tan, and Patrick Lam. 2014. Correlations between bugginess and time-based commit characteristics. *Empirical Software Engineering* 19, 4 (2014), 1009–1039.
[13] Jose Antonio Escobar Garcıa. 2011. Software Development and Collaboration: Version Control Systems and Other Approaches. (2011).
[14] Foutse Khomh, Massimiliano Di Penta, and Yann-Gael Gueheneuc. 2009. An exploratory study of the impact of code smells on software change-proneness. In *Reverse Engineering, 2009. WCRE'09. 16th Working Conference on*. IEEE, 75–84.
[15] Jean-Louis Letouzey and Michel Ilkiewicz. 2012. Managing technical debt with the sqale method. *IEEE software* 29, 6 (2012), 44–51.
[16] Panagiotis Louridas. 2006. Static code analysis. *IEEE Software* 23, 4 (2006), 58–61.
[17] B Mexim and Marouane Kessentini. 2015. An introduction to modern software quality assurance. *Software quality assurance: in large scale and complex software-intensive systems. Morgan Kaufmann, Waltham* (2015), 19–46.
[18] Mavuto M Mukaka. 2012. A guide to appropriate use of correlation coefficient in medical research. *Malawi Medical Journal* 24, 3 (2012), 69–71.
[19] Nachiappan Nagappan, Brendan Murphy, and Victor Basili. 2008. The influence of organizational structure on software quality. In *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on*. IEEE, 521–530.
[20] Martin Pinzger, Nachiappan Nagappan, and Brendan Murphy. 2008. Can developer-module networks predict failures?. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*. ACM, 2–12.
[21] Aniket Potdar and Emad Shihab. 2014. An exploratory study on self-admitted technical debt. In *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*. IEEE, 91–100.
[22] Yilin Qiu, Weiqiang Zhang, Weiqin Zou, Jia Liu, and Qin Liu. 2015. An Empirical Study of Developer Quality. In *Software Quality, Reliability and Security-Companion (QRS-C), 2015 IEEE International Conference on*. IEEE, 202–209.
[23] Foyzur Rahman and Premkumar Devanbu. 2011. Ownership, experience and defects: a fine-grained study of authorship. In *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 491–500.
[24] Eric Raymond. 1999. The cathedral and the bazaar. *Philosophy & Technology* 12, 3 (1999), 23.
[25] Per Runeson, Martin Host, Austen Rainer, and Bjorn Regnell. 2012. *Case study research in software engineering: Guidelines and examples*. John Wiley & Sons.
[26] Diomidis Spinellis. 2005. Version control systems. *IEEE Software* 22, 5 (2005), 108–109.
[27] Richard Stallman. 2016. FLOSS and FOSS. (2016). Retrieved June 26, 2017 from https://www.gnu.org/philosophy/floss-and-foss.en.html
[28] Chris Sterling. 2010. *Managing Software Debt: Building for Inevitable Change*. Addison-Wesley Professional.
[29] Web Technology Surveys. 2017. Usage of operating systems for websites. (2017). https://w3techs.com/technologies/overview/operating_system/all
[30] Web Technology Surveys. 2017. Usage statistics and market share of Apache for websites. (2017). https://w3techs.com/technologies/details/ws-apache/all/all
[31] GNU Operating System. 2017. What is free software? (2017). Retrieved June 26, 2017 from https://www.gnu.org/philosophy/free-sw.html
[32] Antonio Terceiro, Luiz Romario Rios, and Christina Chavez. 2010. An empirical study on the structural complexity introduced by core and peripheral developers in free software projects. In *Software Engineering (SBES), 2010 Brazilian Symposium on*. IEEE, 21–29.
[33] John T Wixted and Ebbe B Ebbesen. 1991. On the form of forgetting. *Psychological science* 2, 6 (1991), 409–415.