UNIVERSITY OF KENT

Are Good Developers More Focused? Data-Mining GitHub

Author: Dorota Oleszczuk
Supervisor: Radu Grigore
Submission Date: September 9, 2018

UNIVERSITY OF KENT

Are Good Developers More Focused? Data-Mining GitHub

Dorota Oleszczuk

Acknowledgments

I am grateful to Dr. Radu Grigore for his support and for sharing his implementations of the GitHub mining code which served as an example for this research. I also thank the DejaVu team for providing additional information and support for the DejaVu tool.

Abstract

The modern-day Software Engineering field requires diverse skills from programmers. With the growth of Open Source Software (OSS), most contributors work on many different technologies throughout their career, often working on multiple technologies each day. However, switching between technologies and coding styles may adversely affect developers' focus. It is important to understand what makes programmers thrive and what can affect their skills negatively. In order to improve their abilities, developers often look up to other established developers. Experience and productivity are some of the traits believed to characterize *good developers* to whom other programmers look up. But what about their focus? To answer the question of whether *good developers* are more focused, I analyzed focus of *good developers* based on the number of extensions they work on within a specified time period.

This paper used a method of evaluating buggy lines in developers' contributions to divide a group of authors into *good* and *bad*. The results are presented in the form of a high-level focus comparison between two groups. I have found that *good developers* are more focused daily and even yearly, but they diversify the technologies they use over their careers. I also investigated whether developers interact differently with selected technologies. *Good developers* focus more than others on nine out of ten most popular extensions. All developers have the same focus level for .rb. The findings can help developers reflect on their programming practices, improve their code quality, plan projects better. An additional research could extend this study with a more fine-grained definition of developers' focus that takes into account temporal switching between technologies. Also, other characteristics that affect the quality of developers could be investigated in order to present a full picture of what makes a *good developer*.

Contents

List of Figures

List of Tables

1. Introduction

"Design and programming are human activities; forget that and all is lost" – Bjarne Stroustrup

Software development is an inherently team-based activity. Empirical analysis can be used to examine development processes and programmers' behaviors. Consequently, it can yield important insights for the development community. Such knowledge can enable software engineering teams to make decisions that will benefit the team and the product. The key to building a successful product is to understand how developers work and what makes them thrive or struggle. Many characteristics have an impact on developer's quality. In this paper, I analyze one of those characteristics to answer the question of whether *good developers* are more focused.

Assessing developers' quality is not easy. There are many criteria to be taken into account, and some of them may be subjective. In this paper, an automated process was used to select *good developers*. *Good developers* were chosen as those who introduce 0% of bugs. It was a harsh criterion based only on the bug-introducing lines and it was not optimal; developers selected as *good* in this paper were not ultimately superior, as well as those selected as *bad* were not ultimately inferior. The terms had to be chosen for the sake of the distinction and ease of describing further research. To analyze focus of *good developers*, authors had to be first divided into two groups— *good* and *bad*. The details of the method used are described in the Methods section followed by explanations of its shortcomings in the Threats to Validity section.

1.1. Overview of Contents

Following the introduction, this paper contains eight main parts. The Background section introduces the research questions and Prior Work discusses related research. The Methods section describes the data gathering and analysis process; it is divided into five subsections. The Dataset and Repository Selection sections talk about the data sources and the project selection criteria. The Good Developers Selection part explains the process of identifying the fix-inducing changes and developer classification criteria. In the Focus Analysis subsection, you will find the focus definition and two ways the focus was measured in this paper. Lastly, the Testing subsection describes the testing and verification done on the scripts and the resulting data. The Threats to Validity section discusses the precision of the measurements used and the known limitations of the research. Results, and Implications for Software Practice sections include a description and a significance of the findings. The Future Research section discusses multiple ways the topic of focus and developers' quality can be studied further. Finally, the Conclusions contain the summary of all the ideas and findings.

2. Background

Since the rise of GitHub's popularity, the abundance of information that became available allowed researchers to examine programmers' qualities and behaviors. As of 2018, GitHub contains over 85M repositories with over 28M contributors [1] and hosts many large OSS projects like Linux Kernel or Microsoft's Typescript. This research uses commits data together with the GitHub issues database to analyze focus of *good developers*.

The question of what constitutes a *good developer* has never been easy to answer and it often sparked disputes due to many characteristics of a *good programmer* being subjective and context-dependent. One of the more obvious ways to measure developer's quality is looking at her bug-introduction ratio. It is logical to expect *good developers* to introduce fewer defects in their code. It is not the ultimate criterion, but it is a decent estimation of developers' quality, which may be assessed based on their committing behavior and the defect database for the project. That was the approach used in multiple previous research works[2, 3, 4, 5], and the one used in this paper. Using this method, I found that the majority of developers were *good*. So what does make *good developers* introduce fewer bugs? In this paper, I examine one of the characteristics believed to affect developers' quality— their focus—in an attempt to answer the question:

Goal: Are *good developers* more focused?

Despite the recent drive towards specialization, developers possess skills in multiple technologies, which encourages them to contribute to many parts of the project they work on, or even to multiple projects. Switching between different technologies and coding styles bears a mental burden that may impair code quality. Focused developers are expected to provide better code[5].

There are many ways to measure a developer's focus. One can analyze switching between files, packages, programming languages, or even specific functions within files. In this research, focus was measured based on the extensions worked on within a period of time. Analyzing focus based on switching between files may be too strict. For example, a developer who worked on multiple files would be described as not focused, even if those files were related and used the same technology. Analysis based on the programming language may be too broad because projects are often written using mainly one programming language; therefore, most programmers would be classified as focused. Analysis of focus based on the extensions, however, is the middle ground between the two. If a developer worked on five .java files during the day, she was described as focused; meanwhile, a developer working on two .java, two .jsp, and an .xml config file, would be characterized by lack of focus.

A focused developer does not have to exclusively stick to a single technology. She may work on a specific technology over a day, a week, or a month, but she may utilize various

technologies throughout her career as a programmer; or she may indeed specialize exclusively in one technology and the high level of focus may be what makes her contribute better quality code. In order to examine the duration of focus for *good developers*, this paper analyzed focus over different time periods:

RQ 1: Are *good developers* more focused all the time or does the focus level differ over various time periods?

To answer RQ1, focus was examined on a daily-, weekly-, and yearly-basis, as well as over a period of four years, which was the full period of the gathered data.

To further understand developers' focus, their daily focus was compared with their weekly focus. Developers' daily focus may be dictated by the work done on previous days—what a developer focuses on on a Tuesday may depend on what she worked on on Monday. On the other hand, the technologies a developer works on each day may be independent. To investigate the behaviors of *good developers*, the following question was examined:

RQ 2: Are developers' daily tasks within each week dependent on each other?

This paper examines focus in two ways. 1) Focus is analyzed *across* technologies, and over different time periods. This approach was covered in RQ1 and RQ2. 2) Focus for *each extension* is examined over active days. The second approach is used to answer the two following research questions. To explore the relationship between developers' focus and specific extensions, this research investigated:

RQ 3: Are *good developers* more focused when working on the most popular extensions?

To answer this question, the paper found the most popular extensions and analyzed focus of each developer group per each popular extension. There are many reasons why developers work on multiple file extensions within a day. It may be dictated by technical dependencies, the developer's interests, or a simple need to work on multiple technologies to increase productivity. If a large difference between the level of focus between *good* and *bad developers* is discovered, it may provide valuable feedback for programming language creators. It could also be an indication that in order to improve code quality, all developers should focus on such technologies. Continuing this idea, the paper explored the question:

RQ4: Which extensions do *good developers* focus on the most?

Here I attempted to discover the technologies that *good developers* focus their attention on the most. The results were compared for *good* and *bad developers* to investigate whether there are any technologies where the difference in focus is the largest.

Understanding how *good developers* work and interact with technologies can be very beneficial for the development community. Empirical Software Engineering is a field

concerned with the analysis of software development practices, and technologies. In this paper, I attempt to contribute to the field of Empirical Software Engineering by answering the question of whether *good developers* are more focused.

3. Prior Work

A great deal of research has been done in the area of Empirical Software Engineering over the last 15 years. Many topics have been examined already; topics like: developers' working rhythm, experience, focus, and productivity; use of Q&A websites like StackOverflow; ownership of code; and their influence on code quality[3, 2, 4, 6, 7, 8, 9]. Numerous research papers studied code quality. In "When Do Changes Induce Fixes?," Sliwerski et al. described an automated method for finding fix-inducing changes. The method was since known as the SZZ algorithms[3]. Sliwerski et al. found that larger changes and modifications made on Fridays tend to contain more defects.

Izquierdo-Cortazar et al. studied the relationship between a developer's experience and bug introduction ratio and they found no correlation[2]; while, in a similar research, Tsunoda et al. found that more experienced developers introduce fewer bugs if the work complexity is kept on the same level[4]. Qiu et al. performed an empirical study of the developer quality metrics and discovered that the quality increases with higher contributions and with software evolution[6]. In "Do time of Day and Developer Experience Affect Commit Bugginess?" Eyolfson et al. studied the relationship of the social characteristics of commits and their "bugginess." They found that late night changes introduce more defects and that developers who commit to a project on a daily basis introduce fewer bugs[7].

Since GitHub became a popular open source version control system, researchers have used it to understand developers' practices and their influence on programming quality. Mo et al. analyzed programmer's commiting behaviors and social network in GitHub to find experts in selected programming languages[8]. In "The Sky is Not the Limit," Vasilescu et al. analyzed the influence of project-level focus on developers' productivity. They found that limited switching between projects is beneficial[9]. In another research, Vasilescu et al. analyzed the relationship between activity on StackOverflow, a Q&A website for developers, and GitHub. They found that more active developers ask less and answer more questions, and that they distribute their code in more uniform ways[10].

The paper most similar to my research is "Dual Ecological Measure of Focus in Software Development" by Posnett et al. In their paper, Posnett et al. looked at the question of whether focused developers produce better-quality code and whether modules that receive narrower focus are of better quality[5]. In their research, the authors used module-based focus measure and found that more focused developers introduce fewer defects, while the modules that receive narrow focus from developers are more likely to contain bugs. In my research, I attempted to find an answer to the question from the opposite side. When one looks at a set of focused developers and a set of *good developers*, the two sets may overlap, but they may not be the same. By analyzing focus of *good developers* instead of the quality of focused developers, I was able to answer the question of whether *good developers* are more focused. Also, in differentiation from Posnett et al., this paper

used extension-based focus, which is described in detail in the Background and Methods sections. Additionally, in my research, I went further to analyze focus over different time periods and to find out how developers interact with each file extension.

4. Methods

In this paper, developers' quality was evaluated using the SZZ algorithm [3] and the programmers were divided into two groups—*good* and *bad*. The focus of both groups was examined for comparison. Focus was measured based on the extensions that developers worked on.

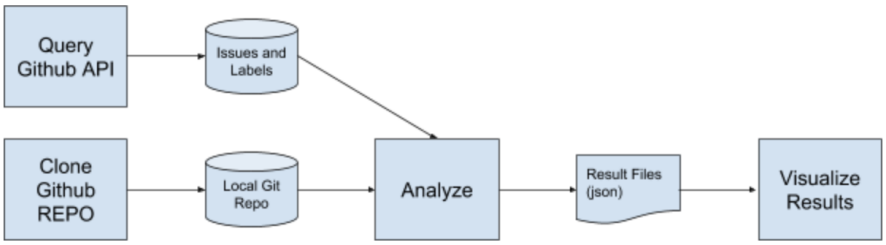

Figure 4.1.: The diagram displays the process followed in this research. The data is gathered via combination of GitHub API and locally cloned repositories. Using the data, good developers are selected and the focus of all the authors is analyzed. The results are saved in a form of JSON files for further visualization and analysis.

4.1. Data Set

The data was extracted from GitHub using a time-filter of four years— from beginning of 2014 until the end of 2017. Over 200,000 commits were collected from ten repositories that contained over 3,000 contributors across seven major languages. The selected repositories included:

- Elasticsearch
- Ansible
- Servo
- Bitcoin
- Selenium
- Spring-boot
- Rust
- TypeScript

- Symfony
- Rails

The issues and the labels were downloaded via the GitHub API, while the commits and further processing was done utilizing additional information obtained by locally cloning the repositories and executing Git commands. The processing is described in detail under Good Developer and Focus Analysis subsections.

4.2. Repository Selection

Because a majority of GitHub repositories is personal and not related to coding[11], the repositories used for the research were carefully selected. The following criteria were used.

Characteristic	Criteria
Age	At least 4 years
Activity Level	Commits: at least 20 per month over the last 4 years.
Amount of Contributors	300+
GitHub Issue Tracking	Yes, the project uses GitHub for tracking issues
Mirror	No, GitHub is the main and only source of code changes
Cloned Code Percentage	0 or confirmed direction indicating that the repository is being cloned from, not to.
Language	Various
Size	Various
Commit Messages Standards	Messages must rely on the conventions for GitHub issue closing like “fixed #123”, “closed #123”, or reference the bug number “#123”

Table 4.1.: Repository Selection Criteria.

The selected repositories were active over four years from 2014 to 2017, and had at least 300 contributors. In order to represent the general developer population better, repositories of different sizes were chosen. Also, the repositories were chosen across multiple programming languages. The repositories are of medium-to-large size, measured by the byte size of files in the repository. This way the research analyzed developers who are more representative of the entirety of the GitHub ecosystem.

Another important criterion was that the repositories had to use the GitHub issue tracking system. Issues data was needed to indicate fixes—the commits that fixed a defect. Mirrored repositories were not accepted.

DejaVu service was used to verify repositories for cloned code. DejaVu allows the verification of the cloned percentage of a GitHub repository [12]. Some of the repositories that were missing in the DejaVu database were still selected in order to provide more language variety. Those repositories were well-established and well-known. The repositories that were not verified for cloned data included TypeScript, Rust, Symfony, Ansible, Servo, and Ruby on Rails. Repositories verified for cloned code either had 0% duplication or were additionally verified for the direction of cloning. If a repository was exclusively cloned from, it could be selected.

4.3. Good Developers Selection

Based on the commiting and issue tracking data from the selected repositories, developers were evaluated and *good developers* were selected.

4.3.1. Process

Good developers were selected first by indicating fix-inducing changes using a set of heuristics based on the SZZ algorithm[3], then by applying developer criteria to the results. First the fixes were identified. Merge commits were excluded from the fix analysis. Only the original commits were inspected. Each merge commit contains metadata about the process of merging the commit A to a target branch and the data of the original commit A. If merges were included in the analysis, it would mean accounting for the commit A twice. The changes that were reverted were also excluded from the research.

The regular expressions used to identify fixes were adjusted to better match the selected repositories and conventions followed in the commit messages. To identify a fix, links were created between a commit and all numbers included in the commit message. Then, those links were rated with two scores— **syntactic** and **semantic**.

Syntactic analysis took into account only syntax. Each of the following criteria increased the syntactic score by one:

- if the number is a bug number or a hash number,
- if the commit message is a plain number or contains a keyword.

Keywords included variation of the words like fix, bug, defect, patch and resolve. If a test was detected, the syntactic score was set to 0. Consider a commit message: "Tests: Add test case from 'fixed #11692'." As the message contains a hash number and a keyword, its syntactic score is 2. Because a bug number 11692 exists and was already fixed, the semantic score is 1. With such results, the commit would be erroneously classified as a fix. After checking for tests, the syntactic score is set to 0, and the commit is not considered a fix. The check for tests prevented false positives of that type.

Semantic analysis referenced the link against the GitHub issue database. It verified whether the number in the link was a bug number. Bugs were identified automatically based on the issue labels. Semantic score increased by one for each of the following criteria:

- if the related issue was marked as fixed or closed,

- If the issue assignee was the author of the commit,
- If the issue title was included in the commit message.

The fix was accepted if semantic score was 1 and syntactic score was greater than 0, or if semantic score was greater than 1. The table 4.2 displays the two conditions.

Condition 1	Condition 2
Semantic score = 1 and syntactic score > 0	Semantic score > 1

Table 4.2.: The table includes two conditions. One of them needs to be met to classify a commit as a fix.

To identify the *fix-inducing changes*, the 'git diff' command was run on the fix to find the modified lines. Then, the 'git blame' command was run on a parent revision of each fix to indicate the commit that inserted the fixed lines. The fix-inducing analysis was performed on a line basis—each blamed line of code was investigated. Each commit within the dataset was annotated with a number of inserted good and fix-inducing lines. Using that data, each developer was marked with the *buggy-to-inserted line ratio* (buggy lines / all inserted lines), which was later used to select good developers. To compare, related papers looked at the buggy commit ratio instead:[7] If commit A inserted 500 lines and among them two lines were buggy, some papers would classify commit A as buggy. Similarly, if 400 out of 500 lines in commit B were defective, the commit B would be described as buggy. It seems inaccurate to treat those two commits in the same way, since there is a clear difference in their quality. Evaluating defects on the commit-basis also results in a higher percentage of defective code than line-based analysis. This research considers line-based analysis preferable.

The dataset was reduced by eliminating non-prolific developers. Non-prolific developers were those who inserted number of lines below the cut-off point. The bottom 35th percentile of lines inserted per repository was selected as the cut-off point. To illustrate an example for one repository, among ten developers having the following total number of inserted lines: 10, 10, 10, 20, 20, 30, 40, 40, 50, 80, the developers with 10 and 20 lines were considered non-prolific and eliminated. Cut-off was set as a percentage per repository because each project varied in size, level of activity, and number of contributors, which made it impractical to set a hard numeric limit.

Some of the authors contributed to more than one repository. In order not to look at a developer who worked on two repositories as two different developers, the authors were combined using a simple email comparison heuristic. The group of good developers was then selected among the combined prolific authors.

4.3.2. Good Developer Criteria

Developers selected as *good* were those with buggy-to-inserted line ratio of 0. Using the criteria, 1,204 of the 2,160 prolific developers were classified as *good*.

4.4. Focus Analysis

Once developers were classified, focus analysis was performed. Focus was measured based on the file extensions worked on by developers in each group. The variety of extensions being modified indicated focus-switching between technologies and coding styles. Focus was measured over different time periods; a developer who worked on few extensions within a time period was classified as focused, while a developer who worked on many extensions was categorized as less focused. Shannon entropy was computed as a measure of focus. Entropy is a measure of uncertainty or disorder[13]. A higher entropy value indicates a higher degree of uncertainty on the developer's next task. With that understanding, higher entropy means that the attention was distributed more evenly across multiple extensions over a time period, and the level of switching between the extensions was higher; therefore, the focus was low. If the entropy was low, there was less switching from one technology to another and the focus was high. In case of this research, Shannon entropy can be also understood as diversity of extensions used. Higher entropy values indicate higher diversity; therefore, less focus. Entropy equal to zero means that a developer worked only on one file extension over the time period. To recall, it is different than working only on one file. A developer could have modified five files during a day, but as long as the files were of the same type (with a single extension), the developer would be considered focused.

4.4.1. Periodic Focus: Fixed Developer-Date Pair

In this part of the analysis, I dealt with three variables: developer, date, and extension. To measure focus, Shannon entropy was computed by fixing a developer and a date variables for different time intervals. Time periods analyzed included:

- 4 years,
- 1 year,
- 1 week,
- 1 day.

Since Shannon Entropy indicates uncertainty in an outcome, it has been a common measure of diversity and focus in many scientific papers[5, 9]. In this research, entropy was defined as:

$$H(A) = -\sum_{i=1}^{n} p_i log_2 p_i$$

where p_i is the probability of an extension i for the specified time period, and n is the total number of unique extensions in the period. For example, to compute daily entropy for a developer working on the following extensions during one day: .java, .java, .py, the calculations are the following:

$$-\frac{2}{3}log_2\frac{2}{3} - \frac{1}{3}log_2\frac{1}{3} = 0.9183$$

For daily focus, each developer had as many entropy numbers as the number of her active days in the dataset. There were up to four yearly entropy numbers per developer because the data was gathered over four years, but some contributors joined the project only for a portion of that time. Lastly, each developer had one overall entropy for the four-year period.

One of Shannon Entropy theorems states that equality between the joint entropy and the sum of individual entropies holds if the variables are independent[14]. By comparing the sum of daily entropies within each week with the weekly entropies, I verified whether developers' focus in a week depends on the previous days, or whether all the days are independent. If developer A's weekly entropy was different than the sum of daily entropies, her daily focus depended on the previous days–what she focused on on Wednesday, depended on her focus on Tuesday. If developer B had weekly entropy very similar to the sum of daily entropies in the week, her days were independent. The similarity threshold was set to 0.1. If the difference between weekly entropy and the sum of daily entropies was less than 0.1, they were considered similar. Otherwise, they were different. Because the weeks containing only one active day could misrepresent the state of focus, they were removed from the dataset for the weekly analysis. Every developer had some weeks with only one active day. For *good developers* it was almost 47% of the cumulative weeks, while for *bad developers* it was only 19% of the weeks.

4.4.2. Focus Per Extension: Fixed Developer-Extension Pair

The analysis from the previous subsection was done by fixing a developer and date. This way, focus per day, week, year, and four years was computed. For this section, I fixed the developer-extension pair, and computed entropy over the active days for a specific extension. This way, I was able to examine whether developers interact with specific extensions in a different way. Using the example from the previous subsection, a developer worked on the following extensions: .java, .java, .py. Her focus for that day was 0.9183 according to the previous calculations. In this section, I looked at how focused was this developer on .java extension, and how focused was she on .py extension. Java entropy for the single day would then be computed as:

$$-\frac{2}{3}log_2\frac{2}{3} = 0.38997$$

Similarly, .py entropy would be:

$$-\frac{1}{3}log_2\frac{1}{3} = 0.52832$$

As you can see, for this method, the probability of the extension is calculated per day, and the entropy is calculated over the active days. Of course, each developer has more than one active day for most of the extensions, so the summation would be done over all active days. Also, most developers work on more than two extensions, so focus would be computed for each extension. This way developer A who worked on 10 unique extensions would have 10 focus scores–one for each extension. Using this method, the focus for ten most popular extensions was analyzed, as well as the extensions the developers focus on the most. The same entropy formula was used:

$$H(A) = -\sum_{i=1}^{n} p_i log_2 p_i$$

where p_i was the probability of the selected extension in a day i, and n was the number of active days. If a developer A's .java entropy was 0, that meant that either on the days she worked on .java, she focused on that extension exclusively; or that she was a prolific developer who did not work on .java files almost at all. In the second scenario, entropy would be 0 because the developer worked on multiple extensions over her days and only a small fraction of them was .java.

In order to find the extensions that developers focused on the most, average entropy was used for the *good* and *bad developer* group. The average was taken over all developers in each group. The extensions with average entropy of below 0.3, and later below 0.6 were considered as those the developers focused on the most. Additionally, usage number condition was added to eliminate the extensions that were used only small amount of times. The file extensions must have been modified at least 100 times to be considered. The usage limit was set low to keep a clear distinction of this analysis from the analysis of the most popular extensions. Additionally, binary and data extensions were excluded from the discussion since developers were not able to write code for them directly. For comparison, average entropy of the opposite group was also calculated for each selected extension.

4.5. Testing

A mix of unit testing and manual tests was used to verify the data. A randomly-selected sample of positively and negatively identified issues was manually checked to verify bugs, taking into account the issue's label, title, and body. Then, randomly selected samples of fix negatives and positives were manually verified against the list of bugs. Manual verification of randomly selected fix-inducing changes was performed on each repository and no anomalies were discovered. Unit tests were written to verify the fix-inducing changes as well as the focus-analysis methods.

5. Threats to Validity

This paper acknowledges a few threats to validity. First, as it was mentioned earlier in the paper, the buggy-to-inserted line ratio is not the ultimate way to judge the quality of developers. The technique does not take into account the soft skills like requirement analysis, communication, time management, or the code characteristics like loose coupling, proper documentation, performance, simplicity, and readability. What is more, the method can only find the introduced bugs that were reported and fixed. If a defect was discovered and fixed immediately without being reported, it would not be taken into account for the analysis. Similarly, if a defect was discovered, but the fix required adding additional logic rather than modifying existing code, the source of such a bug could not be found using the SZZ algorithm[3], and no developer would be blamed for it. Only 10% of fixes inserted new lines instead of modifying existing ones, so the effect on the results should not be significant. The method used in this paper also relies heavily on proper management of repositories. Any bugs not reported, issues fixed but not closed, or fix messages not following the correct syntax standards may lower the precision of the measurements. To mitigate this concern and increase the accuracy of fix-finding method, the repositories used for research were carefully selected, heuristics were extensively tested, and results were verified.

Merge commits pose another threat to validity. Sometimes merging a fix branch to a master branch, creates a merge commit. This happens when the master is not the direct ancestor of the fix branch. Merge commits contain the same file modifications as the parents; therefore, including merge commits in the analysis can result in noting duplicate changes from merged files. For that reason, I excluded merge commits from the analysis. This way no data was lost but the duplicate modifications were avoided.

Similarly, test commits pose a threat to accuracy of the fix-finding method. As manually verified, some of the repositories adopt the convention to add tests for a fixed defect by directly referencing the bug in the commit message. This paper revised a special method to lower the syntactic score for those commits. The suspected test commits were not excluded from analysis the way merge commits were because on occasion some commits could include the test keywords while indeed being fixes, rather than tests. By setting the syntactic score to 0 for suspected test cases, true fixes with strong semantic score were still recognized, while the tests were rejected. The full set of rules recognizing fixes was verified by selecting random samples of fix positives and fix negatives from each repository and manually inspecting the results.

The simple method of merging authors who contributed to multiple repositories could pose additional threat to validity. This paper used the heuristics that merge only developers with the same email address. Therefore, a single author with two different email addresses would be considered as two different authors. The method was chosen because it provides higher precision–no false positives are possible, while recall is not

believed to be significantly impacted.

Cloned code in selected repositories is another recognized limitation. Not all of the repositories were verified for cloned code. Because Dejavu, the cloned code verification tool, does not contain data on all the repositories, not all of them could be verified. If a project contains significant proportion of code that was copied from other repositories, it could misrepresent developers' quality and their focus. To mitigate this concern, the repositories were carefully selected for this project. The repositories that were verified either did not include cloned code, or were only cloned from. The repositories that were not verified for cloned code were well known, which is believed to lower the possibility of contributors cloning code from other projects.

Finally, measuring focus based solely on the diversity of file extensions may not be the optimal method. If a developer worked on six files with .php extension, the method used in the paper would classify her as focused. Comparatively, a developer who worked on six extensions, including two .html, one .js and three .php, was described as not focused. There was no further examination of the modified files. It could be possible that the .php files contained html code and only that part of the files was modified. In that case, the developer was more focused than originally estimated, since five out of six modified files in fact used html technology. Additionally, temporal properties of switching between file extensions could be added to create a more fine-grained measure of focus. This idea is beyond the scope of this research and it is described in more details in the Future Research section.

With the last limitation in mind, this project used extension-based measure as the approximation of focus. One may assume that a developer who worked on one technology during a day was more focused than a developer who worked on five different technologies. Even if the touched files were related, adjusting the mindset to different coding styles lowers developer's focus. It is hard, however, to measure the level of focus for a year because developers work on many technologies during such long period of time. In this research, focus was measured over different, often long, time periods. This is why the main findings are presented in the form of a high-level comparison of focus of two large groups of developers, rather than a fine-grained explanation of each developer's focus. Even when measuring focus over a year or four years, entropy provides accurate estimation to compare two developers groups at a high level.

6. Results

Are *good developers* more focused? Yes, during shorter periods of time.

6.1. Periodic Focus

6.1.1. RQ1: Are good developers more focused over different time periods?

Good developers were more focused daily and even yearly, but not over four years. Over the four-year period, the level of focus was similar for both groups, as presented in Figure 6.1. To recall, the daily focus was measured using all the file extensions during each day per developer; so each developer had as many daily focus values as active days. Similarly, yearly focus was computed using all the file extensions a developer used over a specific year; therefore, each developer had up to four yearly focus values. To compare both groups, this paper looks at the collection of focus values for *good developers* and *bad developers*. Using 1 as the focus threshold, 65% of *good developers*, and 60% of *bad developers* were focused over four years.

Figure 6.1.: The chart depicts the entropy levels for developers over 4 years (2014 - 2017). Over such a long time period, the difference in focus between two groups of developers disappears. This indicates that even though *good developers* tend to focus on a specific technology over shorter periods of time, they do not monopolize their attention overall.

Good developers tended to focus on one task, finish it, and then move on to the next one. They did not stick to working exclusively on one technology over four years, but they focused during limited periods of time, namely daily or yearly. In this paper, the assumption is made that if developers diversified the extensions over four years, they diversified them over longer amount of time as well. This is why the claim is made that

good developers did not monopolize their attention to small set of technologies over their careers. The difference in focus of *good* and *bad developers* was clearly visible on a daily basis, less but still visible on the yearly basis, as depicted in Figure 6.2 and Figure 6.3.

Figure 6.2.: The two charts display daily entropy values for *good* and *bad developers*. The y-axis represents entropy and x-axis represents the developer-date combination. Each vertical line displays en entropy value for a developer for a specific day. It is clearly visible that *good developers* are more focused on the daily basis. The area between entropy level 1 and 2 is mostly filled for *bad developers*, while it is just partially covered for *good developers*, since not many *good developers* reach such a high entropy value. Additionally, much larger percentage of bad developers has entropy beyond 2. Because lower entropy indicates higher focus, it can be deduced from the chart that *good developers* display higher level of daily focus.

6.1.2. RQ2: Are developers' daily tasks within the week dependent on each other?

A majority of developers from both groups scheduled daily tasks in a dependent way during the week. This means that their Friday work was dictated by their focus over the previous days in a week. This part of the research returned similar results for authors from both groups. According to the analysis, for 71% of 2104 good developers, their focus on one day was dictated by work done on the previous days. Only 29% of developers had independent days. Similarly, 66% of 956 bad developers had dependent days, and for 33% of them each day was independent.

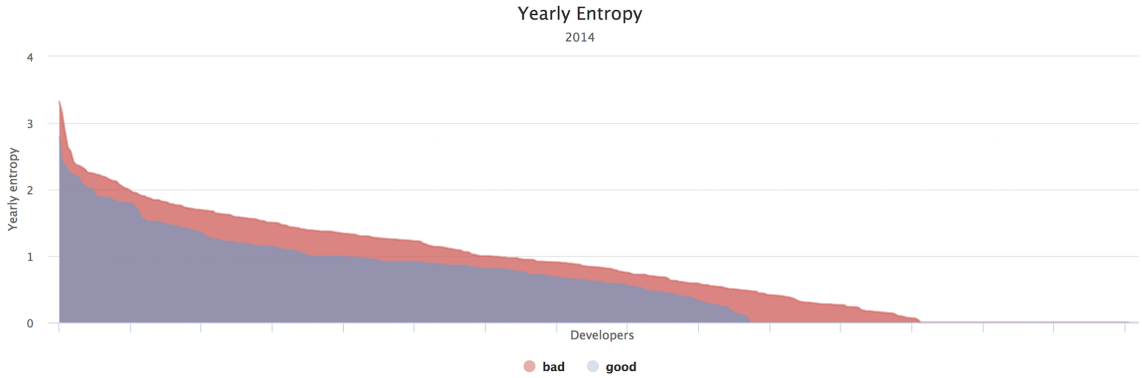

Figure 6.3.: Yearly entropy values for *good* and *bad developers* displayed as overlapping area charts for year 2014. Even on the yearly basis the good developers are more focused, although the difference is much less visible than with daily analysis. The charts for all four years can be found in the appendix.

On average, a *good developer* worked on 3.31 different extensions over the four-year period, while a *bad developer* worked on 6.41– almost twice as many. The next section describes the results of focus analysis per extension.

6.2. Focus per Extension

Generally, entropy numbers per extension for *bad developers* were much higher than for the other group, which indicated lower focus of *bad developers*, as can be seen on Figure 6.4.

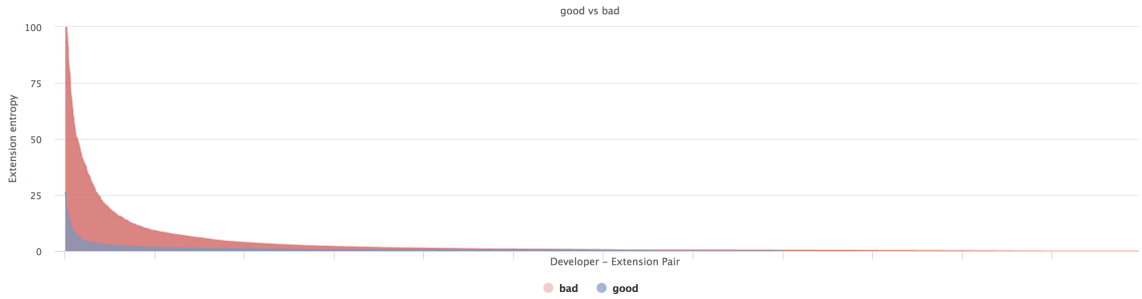

Figure 6.4.: The graph depicts entropy values for the developer-extension pairs. The x-axis contains the developer-extension pair, while y-axis has the entropy value for the pair. The entropy for *bad developers* went up to 188 for a few extreme cases, while the maximum for *good developers* was 25. General higher numbers for *bad developers* suggest that they do not focus on one specific extension, but work on multiple daily. To recall, the highest entropy numbers mean that developer was distributing the focus evenly across the extensions. These results are consistent with the previous analysis.

Entropy values for *bad developers* went up to 188 on the extreme end. Comparatively, a few entropy scores for *good developers* went up to 26 on the extreme end. To recall,

this part of the analysis fixed developer-extension pair to discover developer's focus per extension. Figure 6.5 displays a zoomed-in version of the previous chart to show clearly the difference in values for lower entropy (the long slim part of the graph along the x-axis).

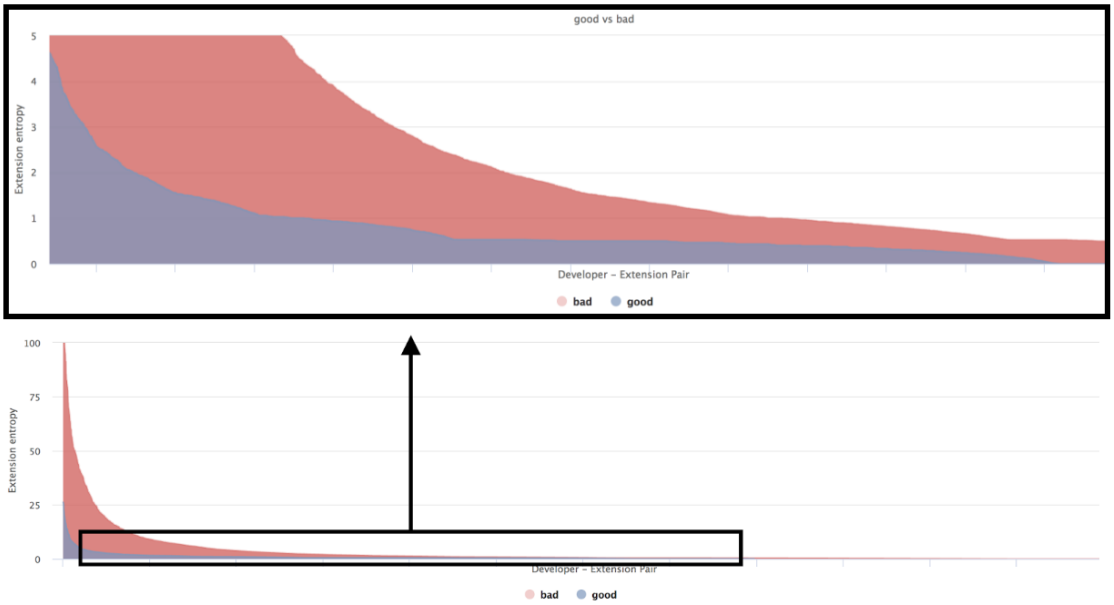

Figure 6.5.: The chart displays a portion of the entropy per extension chart from Figure 6.4 to emphasize the difference between low entropy values of *good* and *bad developers*. The graph clearly shows that entropy values for *bad developers* were higher. For *good developers*, 78.27% of developer-extension pairs have an entropy value below 1. To compare, for *bad developers*, only 8.54% have entropy value below 1.

Entropy values for the majority of *bad developers* were between 0 and 5, with only 8.54% being at or below 1. For *bad developers*, 86.37% of developer-extension pairs were at or below entropy value of 5, and 92.44% were at or below entropy value of 10. Comparatively, a majority of *good developers*' entropy numbers were at or below 1. For *good developers*, 78.27% of developer-extension pairs were at or below entropy value of 1, and 90.82% were at or below entropy value of 2. Entropy ranges and the respective numbers of developer-extension pairs for each group are displayed in Table 6.1 .

6.2.1. RQ3: Are good developers more focused when working on the most popular extensions?

Among the 10 most popular extensions, *good developers* focused on all of them more than *bad developers*. In general, there was a large difference in focus for all popular extensions except .rb. The focus for both groups was similar for the Ruby extension–even *good developers* displayed lower focus. The most popular extensions for the developers in the data set included .java, .js, .json, .lock, .md, .py, .rb, .rs, .txt, and .yml. Some of the extensions are utilized for documentation rather than coding like .txt or .md. Json

Entropy Range	Number of Developer-Extension Pairs	
	Good Developers	*Bad Developers*
0	690	523
0 < entropy <= 1	2430	3229
1 < entropy <= 2	500	834
2 < entropy <= 5	271	706
5 < entropy <= 10	59	372
10 < entropy <= 20	29	226
20 < entropy <= 50	7	167
50 < entropy <= 100	0	61
100 < entropy <= 200	0	9

Table 6.1.: The table displays ranges of entropy values and corresponding number of developer-extension pairs for *good* and *bad developers*. Clearly, majority of pairs for *good developers* is between 0 and 1. To compare, majority of pairs is between 0 and 5 for *bad developers*.

format may be argued to be a data file that should not be used in the analysis, however, a lot of Javascript and TypeScript frameworks use it for configuration and that is why the extension was included in this part of the research. The .lock extension is used by the operating systems to limit concurrent access to resources so it is excluded from discussion. Otherwise, the popular extensions are not surprising considering that those are the extensions of the majority of the languages from the selected repositories, plus documentation and configuration. The popular extensions were chosen from the pool of all the authors, so some of the extensions were used more by one group and some by the other. Figure 6.6 presents developers' focus chart for .java extension. For comparison, Figure 6.7 shows the chart for .rb. There were less *good developers* working on all of the popular extensions except for .rb, .md and .rs. The charts for all the extensions can be seen in the Appendices.

6.2.2. RQ4: Which extensions do good developers focus on the most?

Good developers focus the most on the following file extensions: .bat, .map, .rc, .sql, .policy. The list contains a combination of configuration files, batch scripts and database scripts. The extensions relate to variety of technologies including command line scripting, SQL, Visual Studio, C++, Java, Html, Sass and game interfaces. For each of these extensions, entropy was checked for the opposite group and it was discovered that *bad developers* were less focused on all of them.

To recall, the extensions with average entropy of below 0.3 are considered as those the developers focus on the most. Additionally, binary extensions were excluded from the result list.

It was hard to find the extensions which *bad developers* focus on the most because entropy values were generally much higher than in case of *good developers*. There was no extension that *bad developers* were the most focused on when applying the original

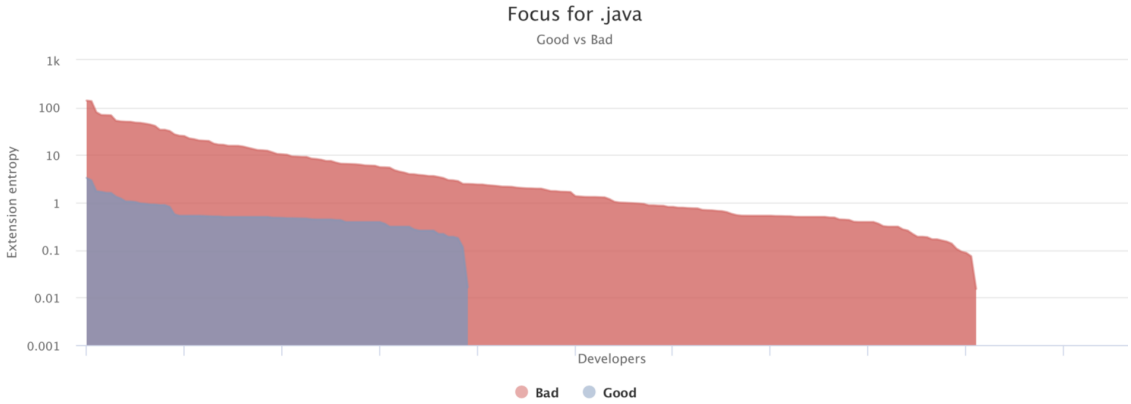

Figure 6.6.: The graph depicts .java entropy levels for developers from both groups using logarithmic y-axis. Clearly, *good developers* are more focused on the Java files. It is easy to perceive that more *bad developers* work on .java extensions than good developers.

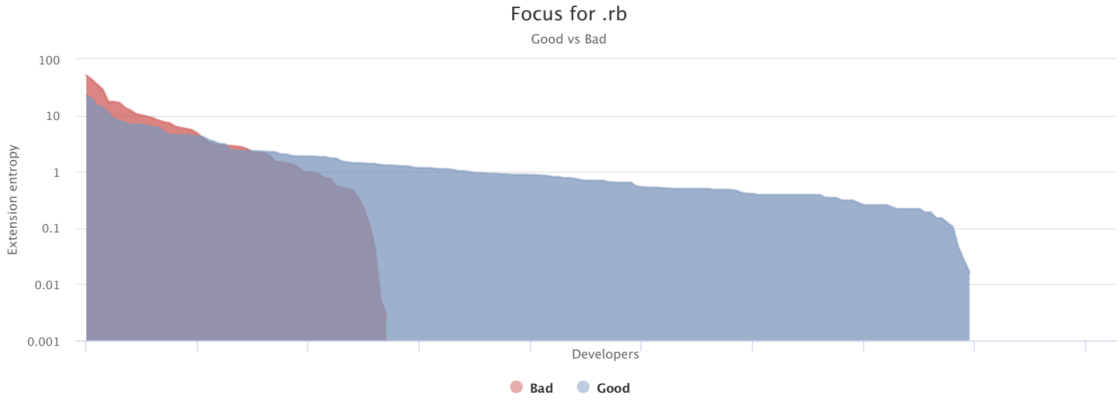

Figure 6.7.: The graph depicts .rb entropy levels for developers from both groups using logarithmic y-axis. The level of focus on Ruby files is very similar for *good* and *bad developers*. Evidently, more *good developers* work on the .rb extension.

criteria. The results are displayed in Table 6.2.

Group	Most focused (entropy < 0.3)	Corresponding extensions entropy from other group
Good	.map 0.261940, .bat 0.239284 .rc 0.278828 .sql 0.283274 .policy 0.245950	
Bad		.map 1.797670 .bat 0.913662 .rc 1.626362 .sql 0.842708 .policy 1.691119

Table 6.2.: The extensions that developers focus on the most (entropy < 0.3). Average entropy across developers in the group was used as a measure. The third column displays the entropy results for the extensions from the other group for comparison. Using 0.3 as the entropy threshold, no extensions were found for *bad developers*.

To further investigate the extensions which *bad developers* focus on the most, additional analysis was done with higher entropy level. Using 0.6 as the entropy threshold, *bad developers* focus the most on the following extensions: .po, a properties file used in Java, and .coffee, a CoffeeScript file that compiles into Javascript. Once the entropy threshold is raised so high, there are numerous extensions that *good developers* focus on. Table 6.3 contains the list of additional file extensions developers focus on the most using the new, higher entropy threshold. That list does not include the extensions enumerated in the previous table. For *good developers*, the extensions are a mix of configuration and build related files and some major file extensions from known programming languages like C, C++, TypeScript, C# or Java.

Developers from two groups differed in their focus per extension. Out of all the extensions, they interacted with .csproj in the most different way, with *good developers* being more focused on it. It should be noted that C# project files can be modified directly by developers, but they can also be automatically changed by an IDE.

Good Developers	*Bad Developers*
.conf 0.3197320635055236	.po 0.5514069703162789
.cc 0.33219280948873625	.coffee 0.48716161553142023
.tsx 0.32548038205462104	
.cs 0.335181806848358	
.po 0.3771562013985646	
.rst 0.38434150516100074	
.java 0.38668939720297063	
.cfg 0.4150984518773275	
.css 0.4331270566825627	
.csproj 0.45364909626365557	
.properties 0.46818206642405324	
.xsd 0.4694220594512273	
.adoc 0.4811941129621413	
.c 0.48835032030656583	
.gradle 0.4981850663754806	
.desc 0.5380845749665812	
.ui 0.5605287497175695	
.htm 0.5729759329981284	
.rake 0.5894723449699876	

Table 6.3.: Additional extensions that developers focus on the most excluding the ones from previous table (0.3 < entropy < 0.6). Average entropy across developers in the group was used as a measure. Clearly *good developers* display significant focus for larger amount of extensions.

7. Implications to the Field

With my research, I seek to contribute to the field of Empirical Software Engineering. It is important to understand the features and practices of *good developers*. Such insights constitute valuable feedback for the community and paths of improvement for the beginner programmers. This knowledge can lead to improving programming and analytics tools like IDEs, or to validating software processes and languages.

In this paper, I studied the question of whether *good developers* are more focused. By examining the commits, I was able to analyze first the programmers' quality, then their focus over different periods of time. In addition, I inspected the extension-specific focus. The research findings include:

- *Good developers* tend to be more focused over shorter time periods but they do not stick to one technology throughout their career;
- A majority of developers schedule their work in a way that each day depends on the previous days;
- Overall, *good developers* display higher focus per extension;
- Among the 10 most popular extensions, Ruby files are those which receive lower amount of focus from both groups;
- *Good developers* focus more on all the most popular extensions except .rb;
- *Good developers* are most focused on the following extensions: .bat, .map, .rc, .sql, .policy; while *bad developers* do not focus on any extension.

Data from the empirical analysis of developers' practices can empower development teams to improve their working habits. Tools like IDEs or version control systems should be able to provide programmers with the focus statistics that can allow developers to distribute their tasks in the most optimal way. Such data can encourage developers to double-check their code if their focus for the day was significantly lower. It can also warn both programmers and managers ahead of time. If the focus level for the day is low, developers can spend the rest of the day focusing more on one technology instead of spreading themselves thin within the project. The focus statistics can also help programmers reflect on their habits and improve over time. It can be also used as an aid to plan daily tasks.

Additionally, information about developers' focus and quality provides useful feedback for project managers. The data may be used to plan maintenance efforts and allocate developers resources within the project. The allocation can be done based on task complexity and the technology involved. Each developer can be assigned to the technology they focused on the most in the past, and to the tasks with proper difficulty based on the past code quality.

8. Future Research

The findings in this research showed clearly that *good developers* tended to focus over a shorter amount of time but, overall, they diversified the technologies they worked with. Additionally, *good developers* were more focused on all the popular extensions except for .rb, where the focus level was similar for all programmers. These findings provide details that facilitate the understanding of the processes used by developers. My research can be further expanded by performing focus analyses that examine temporal focus switching. In this paper, the developer who worked exclusively on .java extensions was described as focused, while the developer who worked on three .java, two .py, and .config was not focused. The order of focus-switching between the technologies was not taken into account. The next step to evaluate focus level more accurately is to analyze the time and order of the switching. For example, if both developer A and developer B worked on six files (.java, .java, .java, .py, .py, .config), but developer A worked first on three .java files, then on two .py files, and last on .config, while developer B worked on .java, .py, .java, .config, .py, .java in that order, then the focus level of the two developers was different. Developer A seems more focused than developer B.

Additionally, a survey method could be employed to investigate the developers' perception of their focus and the reasons why they focus on different technologies. There are many reasons why a developer would switch her attention between different extensions—it could be dictated by technological dependencies. For example, if a developer worked on an .html file, she would likely also work on a .css or .js file. The switch could also be caused by a resource delay. For example, if a front-end developer needed to wait for a back-end developer to finish an endpoint before she could integrate it into the UI. Then, she would switch to different tasks utilizing different technologies in the meantime. Similarly, a developer could be waiting for the tests or deployment to finish and in the meantime she worked on another part of the project. Finally, certain changes just require modifications within files with multiple extensions. It is important to understand not only how developers behave but also the reasons behind their practices.

Finally, there are many other characteristics that could be explored for their relation to developers' quality. Additional study could be performed to find out whether *good developers* are more consistent. One could suspect that *good developers* write code regularly. Possibly, they follow a schedule that optimizes their focus and mental abilities, like working only for six hours a day. It is an interesting idea to explore developers' work scheduling further.

9. Conclusions

This research analyzed focus of *good developers* based on the file extensions used in their work. It was discovered that *good developers* focus more during shorter periods of time, but they diversify the technologies they use throughout their careers. Additionally, this paper showed that *good developers* focus more when working on all of the most popular extensions; except for Ruby files, which received smaller amount of focus from all developers. These findings can be used by developers for self-reflection and improvement of their working habits and development process. This type of analysis could also be incorporated into the development tools to allow programmers make data-driven decision easier. The ways in which developers interact with specific extensions can be researched further for the purpose of discovering which technologies require the least of programmers' attention, and to collect authors' reasons for switching their focus. Additionally, temporal focus switching analysis can be used to explain developers' focus over shorter time periods in more detail. All in all, the findings prove that focus is clearly one of the characteristics that make *good developers*.

Bibliography

[1] *GitHub Facts*. https://github.com/about/facts. Accessed: 2018-08-30.

[2] D. Izquierdo-Cortázar, G. Robles, and J. M. González-Barahona. "Do More Experienced Developers Introduce Fewer Bugs?" In: *Open Source Systems: Long-Term Sustainability*. Ed. by I. Hammouda, B. Lundell, T. Mikkonen, and W. Scacchi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 268–273. ISBN: 978-3-642-33442-9.

[3] J. Śliwerski, T. Zimmermann, and A. Zeller. "When Do Changes Induce Fixes?" In: *SIGSOFT Softw. Eng. Notes* 30.4 (May 2005), pp. 1–5. ISSN: 0163-5948. DOI: 10.1145/1082983.1083147. URL: http://doi.acm.org/10.1145/1082983.1083147.

[4] T. Tsunoda, H. Washizaki, Y. Fukazawa, S. Inoue, Y. Hanai, and M. Kanazawa. "Evaluating the work of experienced and inexperienced developers considering work difficulty in sotware development". In: *2017 18th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*. June 2017, pp. 161–166. DOI: 10.1109/SNPD.2017.8022717.

[5] D. Posnett, R. D'Souza, P. Devanbu, and V. Filkov. "Dual ecological measures of focus in software development". In: *2013 35th International Conference on Software Engineering (ICSE)*. May 2013, pp. 452–461. DOI: 10.1109/ICSE.2013.6606591.

[6] Y. Qiu, W. Zhang, W. Zou, J. Liu, and Q. Liu. "An Empirical Study of Developer Quality". In: *2015 IEEE International Conference on Software Quality, Reliability and Security - Companion*. Aug. 2015, pp. 202–209. DOI: 10.1109/QRS-C.2015.33.

[7] J. Eyolfson, L. Tan, and P. Lam. "Do Time of Day and Developer Experience Affect Commit Bugginess?" In: *Proceedings of the 8th Working Conference on Mining Software Repositories*. MSR '11. Waikiki, Honolulu, HI, USA: ACM, 2011, pp. 153–162. ISBN: 978-1-4503-0574-7. DOI: 10.1145/1985441.1985464. URL: http://doi.acm.org/10.1145/1985441.1985464.

[8] W. Mo, B. Shen, Y. He, and H. Zhong. "GEMiner: Mining Social and Programming Behaviors to Identify Experts in Github". In: *Proceedings of the 7th Asia-Pacific Symposium on Internetware*. Internetware '15. Wuhan, China: ACM, 2015, pp. 93–101. ISBN: 978-1-4503-3641-3. DOI: 10.1145/2875913.2875924. URL: http://doi.acm.org/10.1145/2875913.2875924.

[9] B. Vasilescu, K. Blincoe, Q. Xuan, C. Casalnuovo, D. Damian, P. Devanbu, and V. Filkov. "The Sky is Not the Limit: Multitasking Across GitHub Projects". In: *Proceedings of the 38th International Conference on Software Engineering*. ICSE '16. Austin, Texas: ACM, 2016, pp. 994–1005. ISBN: 978-1-4503-3900-1. DOI: 10.1145/2884781.2884875. URL: http://doi.acm.org/10.1145/2884781.2884875.

[10] B. Vasilescu, V. Filkov, and A. Serebrenik. "StackOverflow and GitHub: Associations Between Software Development and Crowdsourced Knowledge". In: *Proceedings of the 2013 International Conference on Social Computing*. SOCIALCOM '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 188–195. ISBN: 978-0-7695-5137-1. DOI: 10.1109/SocialCom.2013.35. URL: http://dx.doi.org/10.1109/SocialCom.2013.35.

[11] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian. "The Promises and Perils of Mining GitHub". In: *Proceedings of the 11th Working Conference on Mining Software Repositories*. MSR 2014. Hyderabad, India: ACM, 2014, pp. 92–101. ISBN: 978-1-4503-2863-0. DOI: 10.1145/2597073.2597074. URL: http://doi.acm.org/10.1145/2597073.2597074.

[12] C. V. Lopes, P. Maj, P. Martins, V. Saini, D. Yang, J. Zitny, H. Sajnani, and J. Vitek. "DéJàVu: A Map of Code Duplicates on GitHub". In: *Proc. ACM Program. Lang.* 1.OOPSLA (Oct. 2017), 84:1–84:28. ISSN: 2475-1421. DOI: 10.1145/3133908. URL: http://doi.acm.org/10.1145/3133908.

[13] A. Besenyei. *On some subadditivity inequalities of entropies*. http://abesenyei.web.elte.hu/publications/entropy.pdf. 2014.

[14] D. P. Ádám Besenyei. *Linear Algebra and its Applications*. ELSEVIER, 2013.

A. Appendices

Appendices include additional graphs presenting the results of focus analysis.

A.1. Yearly Focus

This section presents four charts for yearly focus (Figure A.1, Figure A.2). It is visible that *good developers* display higher yearly focus than *bad developers*. However, the difference is quite small and it decreases with years.

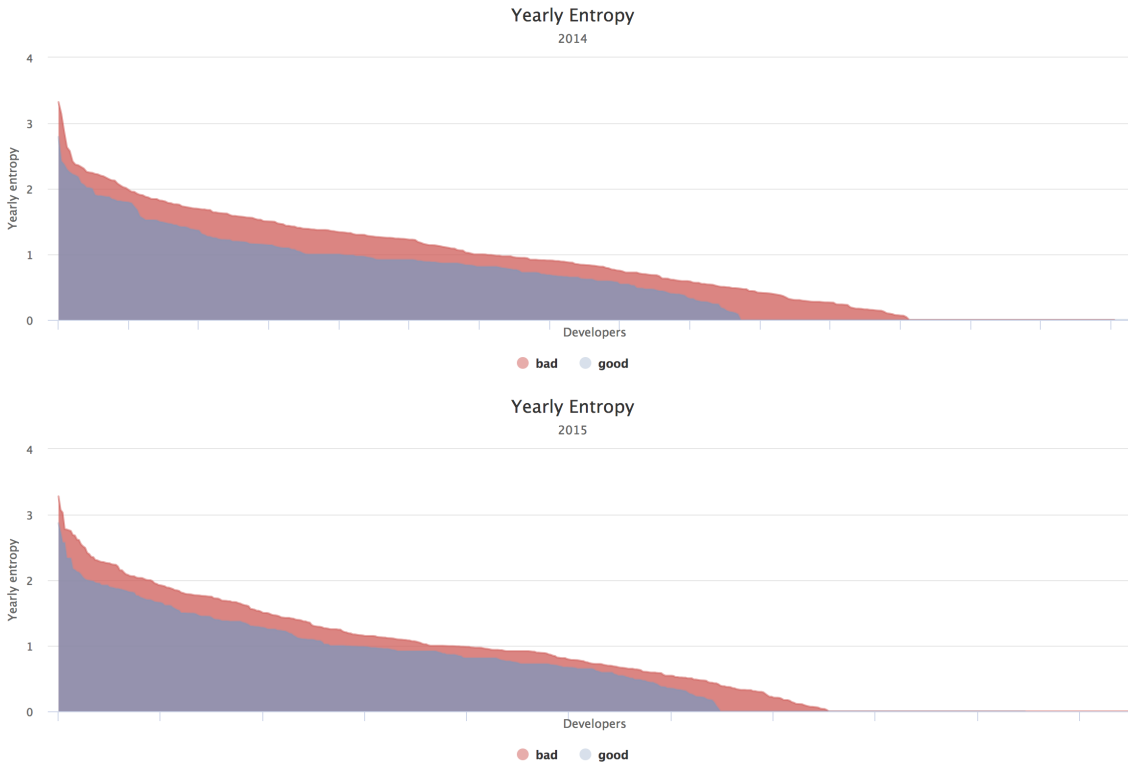

Figure A.1.: Yearly entropy values for *good* and *bad developers* displayed as overlapping area charts for years 2014 and 2015. Even on the yearly basis the good developers are more focused, although the difference is much less visible than with daily analysis.

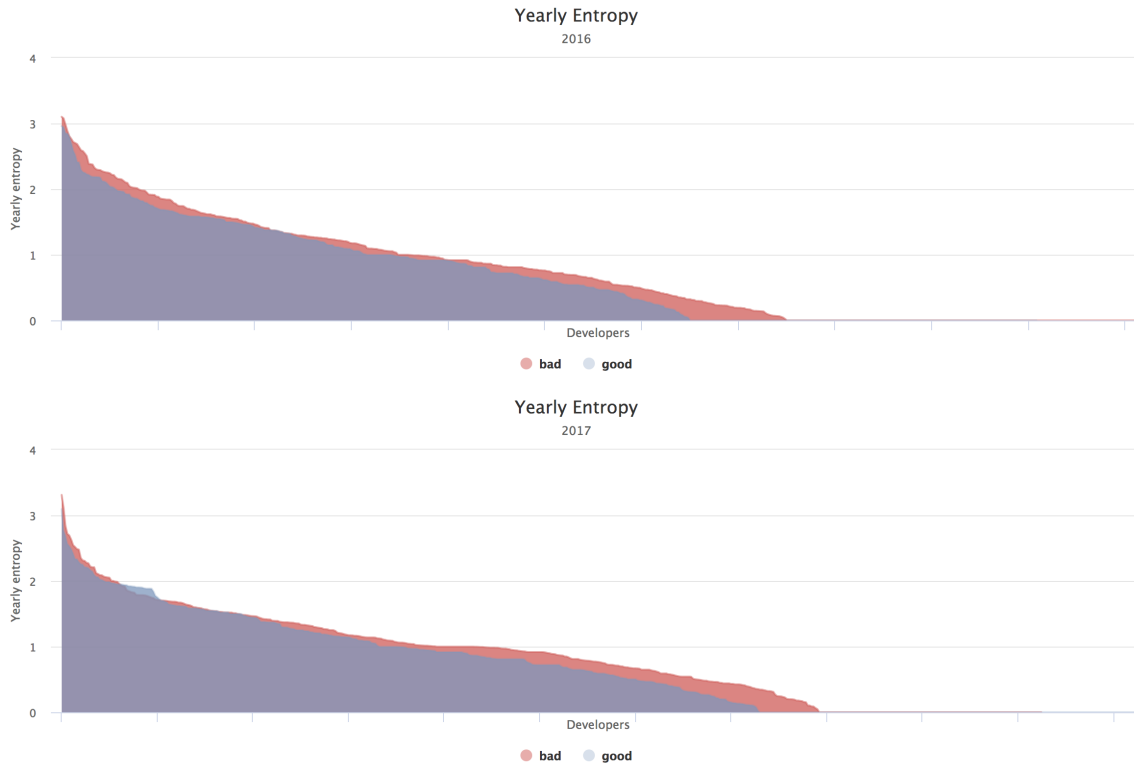

Figure A.2.: Yearly entropy values for *good* and *bad developers* displayed as overlapping area charts for years 2016 and 2017. Even on the yearly basis the good developers are more focused, although the difference is much less visible than with daily analysis. It is visible that the difference between *good* and *bad developers* gets smaller over years.

A.2. Focus For The Most Popular Extensions

This section contains charts presenting the focus for each of ten most popular extensions. It is visible that *good developers* are more focused on all but one extension. Ruby files seem to receive the same amount of focus from *good* and *bad developers* alike. Most of the extensions are related to coding, besides .txt, .md which are related to documentation and .lock which is a program file that developers do not write directly into.

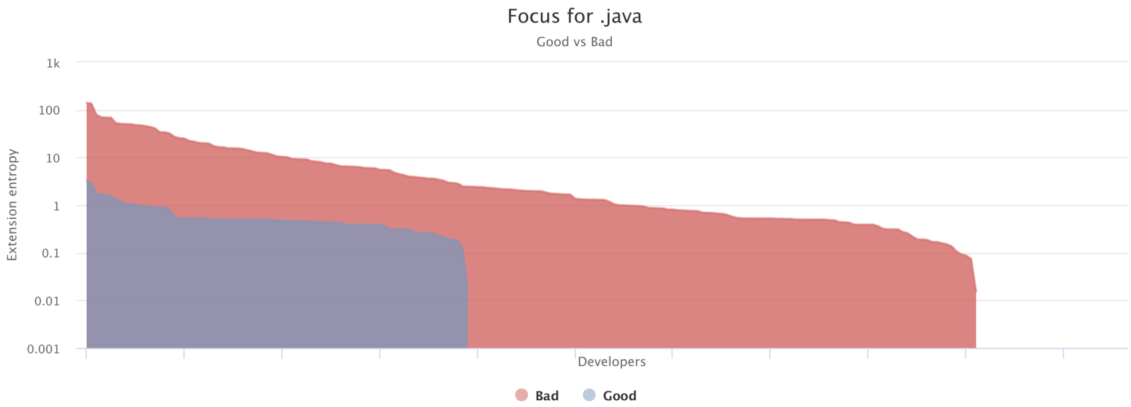

Figure A.3.: Clearly *good developers* focus more on .java extensions since the entropy levels are lower. This chart also shows that there are more *bad developers* working on this extension.

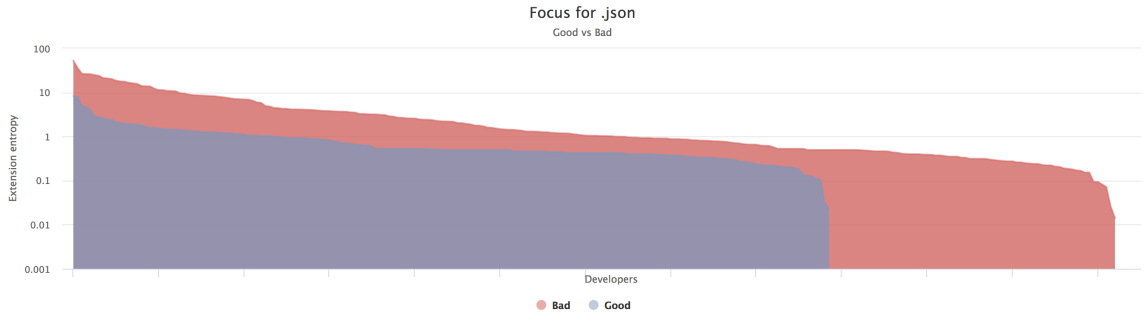

Figure A.4.: Clearly *good developers* focus more on .json extensions since the entropy levels are lower. This chart also shows that there are more *bad developers* working on this extension.

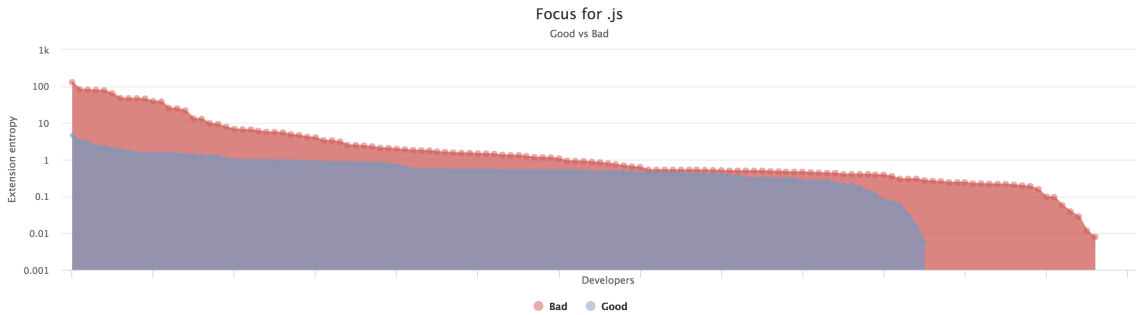

Figure A.5.: Clearly *good developers* focus more on .js extensions since the entropy levels are lower. This chart also shows that there are more *bad developers* working on this extension.

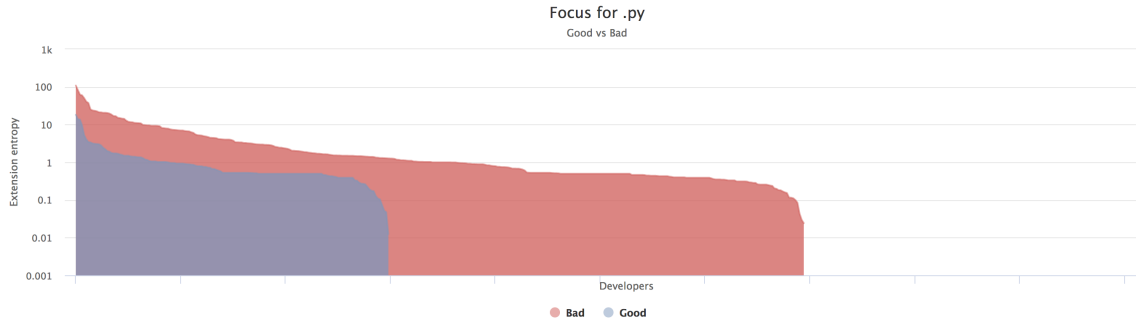

Figure A.6.: Clearly *good developers* focus more on .py extensions since the entropy levels are lower. This chart also shows that there are more *bad developers* working on this extension.

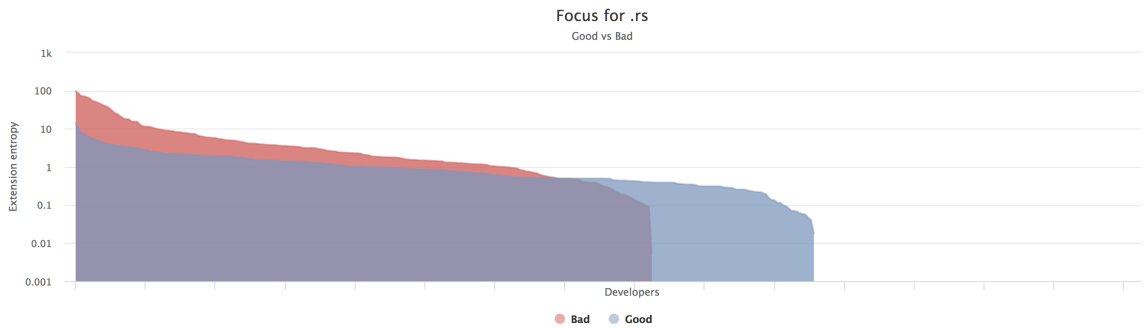

Figure A.7.: Clearly *good developers* focus more on .rs extensions since the entropy levels are lower. This chart also shows that there are more *good developers* working on this extension.

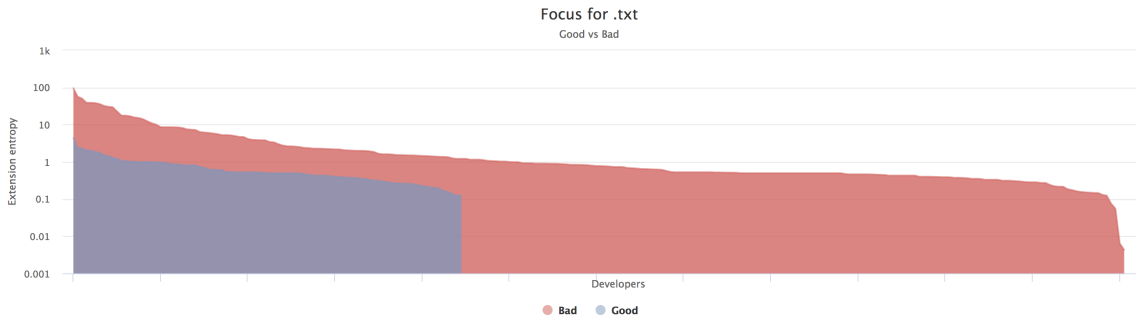

Figure A.8.: Clearly *good developers* focus more on .txt extensions since the entropy levels are lower. This chart also shows that there are more *bad developers* working on this extension.

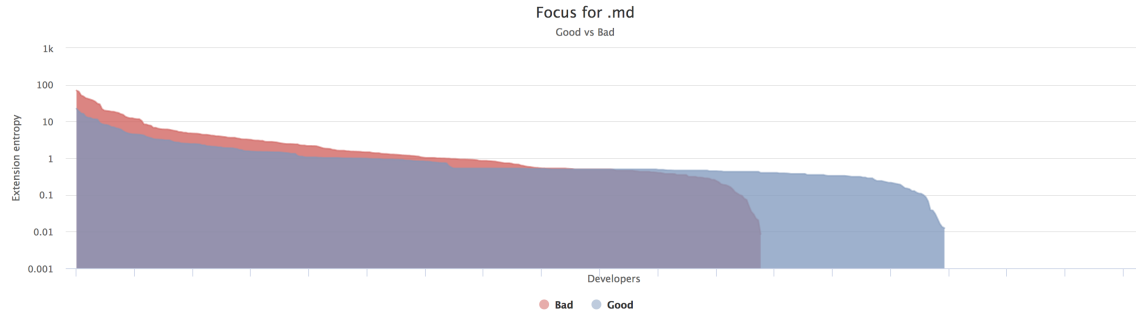

Figure A.9.: Clearly *good developers* focus more on .md extensions since the entropy levels are lower. This chart also shows that there are more *good developers* working on this extension.

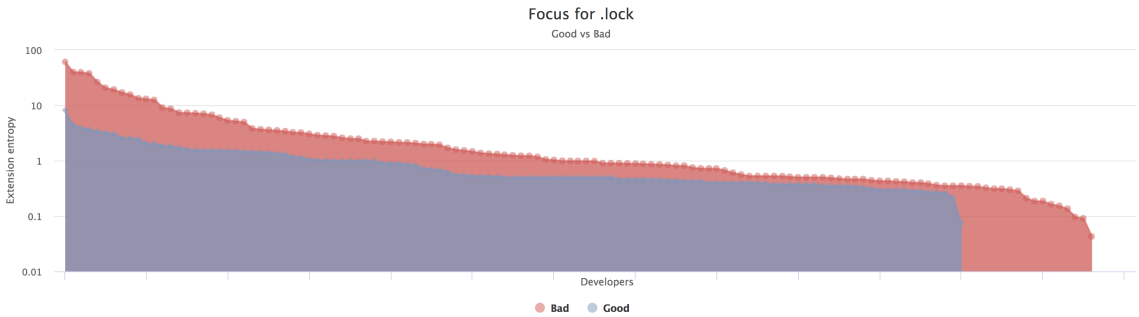

Figure A.10.: This chart is displayed because .lock is one of the ten most popular extensions. However since developers do not directly modify it, it is not discussed.

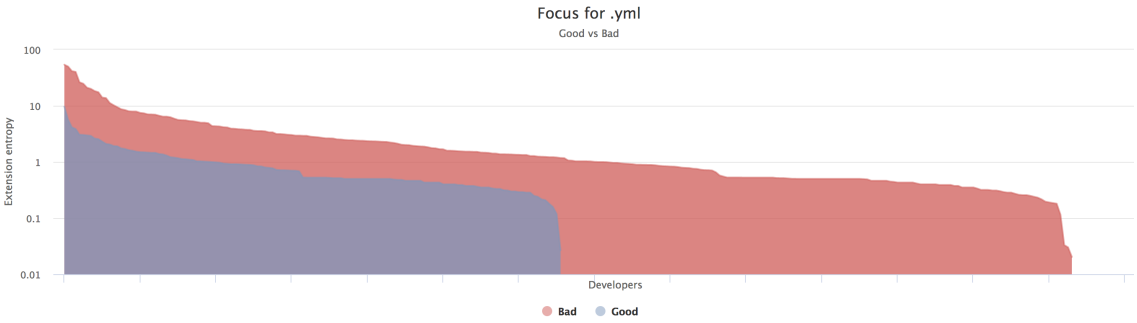

Figure A.11.: Clearly *good developers* focus more on .yml extensions since the entropy levels are lower. This chart also shows that there are more *bad developers* working on this extension.

Figure A.12.: Both groups of developers focus on .rb extensions in a similar way. This chart also shows that there are more *good developers* working on this extension.