5-2020

Conceptions of Refactoring: An Investigation of Stack Overflow Posts

Steven David Simmons
sds9278@rit.edu

Conceptions of Refactoring: An Investigation of Stack Overflow Posts

by

Steven David Simmons

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of
Master of Science in Software Engineering

Supervised by
Dr. Mohamed Wiem Mkaouer

Department of Software Engineering
B. Thomas Golisano College of Computing and Information Sciences
Rochester Institute of Technology
Rochester, New York

May 2020

The thesis “Conceptions of Refactoring: An Investigation of Stack Overflow Posts” by Steven David Simmons has been examined and approved by the following Examination Committee:

Mohamed Wiem Mkaouer

Dr. Mohamed Wiem Mkaouer
Assistant Professor, RIT
Thesis Committee Chair

Christian Newman

Dr. Christian Newman
Assistant Professor, RIT

J. Scott Hawker

Dr. J. Scott Hawker
Associate Professor
Graduate Program Director, RIT

Abstract

Conceptions of Refactoring: An Investigation of Stack Overflow Posts

Steven David Simmons

Supervising Professor: Dr. Mohamed Wiem Mkaouer

Refactoring is a common activity in software development. Developers make changes in the code in order to achieve a desired effect such as better performance, conformance to new business rules, or the removal of code anti-patterns such as code smells. However, refactoring operations often fail to achieve the results that are expected of them. There have been many studies conducted to assess their impact and effectiveness in different scenarios, but they have not returned consistent results. At times, studies have even shown decreasing and increasing in code quality from the same operation. This study investigated whether a common set of well-known refactoring actions defined by Fowler aligned with the terms and actions discussed by developers in the process of refactoring. It obtained these scenarios by looking into Stack Overflow posts discussing refactoring to see what actions and goals users were pursing. We hypothesis that there may be discrepancies between the actions discussed and the context (i.e., web-development vs. database) or technology (i.e., Java, PHP, Python) where different refactorings are more easily implemented. It was found that the number of identifiable refactoring scenarios increases when the scenario contains matching components described in the methods (i.e., Extract Method, Rename Class, etc.). Additionally, developers often only have a vague conception of what actions they believe will achieve the goals of their refactor. The conclusion drawn from these results is

that refactoring suggestions must be aware of the context they will be applied to in order to align with the developer's expectations. These methods also must be explicitly aligned with specific quality improvements or changes in order for developers to feel more comfortable using them to communicate their refactoring intentions.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Refactoring is a popular process for improving the internal structure of the code [1]. The process is often done in order to satisfy evolving requirements [27], add new functionality or remove bugs in the code [37], or to remove design anti-patterns or ”code smells” [1]. Code smells are structural problems in the program with common examples including *God Class*, *Feature Envy*, and *Long Method* [1]. The result of refactoring is a program that is measurably improved, usually in an improvement of quality metrics or in the removal of an undesirable piece of code, such as a code smell.

Refactoring is a consistent practice that spans several domains such as mobile, desktop, and embedded software and is independent of any particular programming language. Several metrics have been proposed in order to measure the impact of refactoring and identify refactoring opportunities such as the CK-Metrics [11] and the MOOD metric suite [17], and others are consistently employed in order to identify areas for refactoring opportunities, particularly in implementations of refactoring tools such as Ref-Finder [32].

However, performing refactoring does not reliably achieve improvements in software quality or the removal of code smells. There have been several studies that have attempted to leverage metrics in order to measure the results in refactoring [10, 2, 4, 30, 13] but they all commonly attest that there is not a shared consensus amongst their results when refactoring is attempted. Refactoring operations taken to remove code smells are often unsuccessful and in fact instead have a significant change to introduce new smells into the code [9]. It is still currently unclear what factors are contributing to this discrepancy between the application of refactoring and the lack of consistent benefits of those actions

in the literature.

This study seeks to begin an investigation into how accurately refactoring methods are being applied by performing a study analyzing Stack Overflow, a popular Q&A site where programmers ask questions. Refactoring is often a stated action or goal in making changes to the code, therefore this study will attempt to identify trends in the posts made when developers attempt to refactor their code. Identifying what developers perceive the intended goal of their refactoring will be along with the context (i.e., language, domain, etc.) in which they make that assumption will allow us to form a conception of what refactoring is and hopes to achieve in these scenarios. This will allow us to see if there are common scenarios where specific refactoring actions are applied or if there is a mismatch between the actions commonly suggested in literature and what actions are suggested within the discussion. We will also seek to identify the reasoning that developers undertake these actions and see if there is any consistency in their mindset when they undertake refactoring actions.

Our ultimate goal in this is to identify a basis as to why refactoring does not attain the results expected of it and highlight any factors that may contribute to changing the context in which the refactoring is occurring. Through this, we believe refactoring can become a more accurate activity that better acknowledges factors which require it to adapt its methods to create more effective, desirable change.

Chapter 2

Related Work

2.1 Ineffectiveness of Refactoring

This thesis was initially inspired by the work done by Bavota et. al. on identifying the relationship between quality and refactoring [2]. They noticed a lack of studies that quantitatively analyzed which quality characteristics in the code are more likely to be subject to refactoring. This also implied that there was a lack of evidence that the presence of concerning quality metrics (i.e., code smells) actually prompted the developers to refactor the code. Analyzing 63 releases of 3 large Java programs (Apache Ant, ArgoUML, Xerces) with *Ref-Finder* [32], a tool that automatically detected 52 types of refactoring methods, they uncovered 12,922 refactoring methods present in the systems. Of those observed to be directly acting on files exhibiting code smells, only 7% of total operations actually removed the smell. The reasoning for this discrepancy was interpreted by leveraging a previous study done by the authors about developer perception of code smells [29]. There it was considered that only particularly severe code smells were considered worthwhile to refactor. An additional interpretation was that refactoring actions were only mitigating the smell without fully removing it, although it is unclear from this study of that was the developer's intent when the change was made. The interpretation we take from these results is that refactoring, which often has a main goal of removing smells, does not often achieve the results that it is assumed to be able to meet. Therefore, this study attempts to identify what developers seek to accomplish in each refactoring opportunity through an investigation of Stack Overflow posts. Bavota's initial study was followed up by couple [10, 9] longitudinal

studies that expanded upon this research, addressing some of its limitations such as its very small sample size (only the major releases of 3 projects were looked in the original study), by expanding it to over 20 open source projects for each study and examining each version.

A longitudinal study was performed by Cedrim et. al in order to identify the effects of refactoring on code smells [9] after observing a lack of studies categorizing the positive or negative effects of refactoring methods on smells. The study was conducted on 23 GitHub projects that were highly popular (as measured by their stars), had active issue tracking systems, and that had a majority of files (above 90%) written in Java. Using a tool called *Refactoring Miner* [39, 40] to identify the refactorings in the code, the authors considered refactorings to be either positive, negative, or neutral based on the amount of code smells the change had increased, decreased, or remained the same. Code smells were detected using three sets of metric thresholds in order to cover for each threshold potentially having its own results. These thresholds were based on previous studies by Macia et al. [26] and Bavota et. al [2] as well as including a relaxed set which was more inclusive. They identified 16,566 refactorings in the code set with only 79.4% of the operations directly touching a smelly element. Only 9.7% of the operations actually removed the smell, with 33.3% inducing at least one additional smell, and the remaining 57% not affecting the number of smells in the code at all. This suggests that even having previous studies prove that developers detect smells in a similar method to the one used by the study to detected smells [19, 15], they are still not accurately able to accurately able to target and remove the smells. This study seeks to confirm that developer expectation of refactoring actions are in line with their actual effect to see if that helps explains these results.

Tahir et. al performed an analysis of how developers discussed code smells on Stack Overflow[38]. Based on other recent work in the field [29, 45] and the increase in reports that smell detection tools were often returning false positives [16], they postulated that the software community was facing a gap in their understanding of the criticality of code smells and anti-patterns. Taking a set of question posts identified by code smell keywords, they determined which were the most actively discussed by utilizing the post score and

view count. Selecting the top 100 posts from this set, they were able to determine the common corrective action suggested to refactor the smell. Of the minority of answers seen that offered a fix (25%), they did not specify any specific name to the operations (such as a refactoring method) and often were given in code snippets. Our study expands this perception analysis to refactoring while also attempting to identify the context in which the developers are refactoring.

2.2 Misconception in Software

Pantiuchina et. al [30] attempted to perform an investigation into if quality metrics are able to capture code improvements as perceived by developers. Using the GitHub Archive, a project recording public GitHub events in a JSON format, they looked for any commits that mentioned one of the quality metrics of interest: coupling, complexity, cohesion, readability. After filtering out any commits that testing related commits or any commits that affected more than 5 files, the authors manually reviewed the remaining commits to ensure the main goal of the commit was quality improvement. This process resulted in 1,282 commits from 986 Java programs on GitHub. A record of the quality metrics of the code (e.g. LCOM, WFC, RFC) was taken before the commits in the set and compared against the metrics present after the commit was applied. The results found that there was often inconsistencies between the changes in metrics and the developer's perception of improvement in the code. There were several instances observed where a developer undertook refactoring in order to improve an aspect of the code (e.g. reducing complexity by extracting several method from a long method) resulted in a negative change in the metrics for the affected files. Our study takes a similar assumption to the conclusions found in this paper that a developer may have a different conception of a refactor than is suggested measurements or static analysis tools. Therefore, it is imperative to identify if there are common trends in developers' perception to see if they can be greater alignment and support provided by those tools.

Zapalowski et. al performed a study investigating six systems to determine what sort

of divergence happens between conceptual architecture design and actually implemented design [47]. Despite the importance of software architecture, many systems do not have reliable architecture documentation which dictates the rules different components should follow (i.e. when methods and functions can be invoked and by whom). This lack of documentation contributes to architecture erosion (i.e. creation of undesired dependencies among modules) and may lead to the loss of knowledge on the original conception of the system as it slowly drifts from its intended implementation. Their analysis revealed that violations of architecture rules were detected in all the systems, ranging from 44.4% - 73.9% of all dependencies in the observed system. Furthermore, there was a lack of dependencies allowed by the rules with the highest observed system having 39.7% of its dependencies unimplemented. This implied that the rules in the system were too loose and potentially allowing unnecessary or harmful dependencies to be formed within the system. While this study does not focus on refactoring it does reinforce our assumption that there are software concepts that can diverge from what they set out to and have potentially harmful impacts on the software.

2.3 Stack Overflow Studies

Zhang et. al looked into the reliability of code snippets on Stack Overflow in terms of correct API usage [48]. They determined the proper API usage patterns by mining code examples from GitHub using a program they developed called *ExampleCheck*. After obtaining these patterns, they select posts that contained an API from list of 100 Java APIs that were discovered by parsing the Stack Overflow dump taken on Oct. 2016 and an API misuse benchmark tool called MUBench [6] and extract their code snippets. Out of the 217,818 posts identified, they found that 31% had potential misuse of their APIs with consequences such as resource leaks, incomplete action (i.e. not completing a transaction after modifying data), or potentially crashing the program. They also found that the highly voted posts are not necessarily correlated with fewer API usage violations. As previous studies have shown, the score is more closely correlated to the presence of code examples and

detailed step-by-step instructions in the post [28]. The vote score they used in their study is the same score value used in our popularity calculation, so the insights they found on highly scored posts have the potential to translate to different domains, such as refactoring in our study.

Imtiaz et. al performed a study on Stack Overflow where [20]. This study also used the MSR 2019 challenge dataset that this study uses. They built their set based on a list of popular SATs from OWASP(Open Web Application Security Project), forming a set of 17 keywords that were searched for in the tags of question posts. After those posts were identified, they were further filtered by 9 keywords related to alerts. The authors found that the posts had three common themes: Ignore/Filter alerts, False Positive Validation, and Handling False Positives, accounting for over 90% (93.5%) of the posts observed. The alerts the posts described were categorized according to a grouping scheme proposed by Johnson et. al into two categories that are comprised of 10 challenges developers faced [**?**]: *Knowledge Gap*, a gap between the developer's knowledge and the information performed by the alert and *Knowledge Mismatch*, a mismatch between what the developer expects the alert to communicate and what it actually means. This study is similar in that is trying to understand a consensus in how developers handle tool alerts while ours is focused on refactoring. The Knowledge Mismatch category reinforces our assumption that there might be areas that have a gap between what the developers perceived and what is actually occurring.

Chapter 3

Methodology

In this chapter, we will explain the research questions we formed for this study. These questions will provide a framework for our understanding and explain the actions we will take in this study. Answering these questions will place us in a better position to understand if the actions developers discuss while refactoring and the reasoning they have for their approaches. Finally, we will explain how we have gathered the data we used for the study and in what ways we have processed it. We will also explain how the result of each step of the processing will assist in answering the research questions.

3.1 Research Questions

This section will cover the research questions for this study and explain how answering them will further our understanding of how developers interpret refactoring and the actions they take in the name of refactoring.

RQ1: What are the general areas where refactoring is considered unreliable?

This question will help us confirm if the idea that refactoring methods are unreliable is prevalent, at least amongst academia. There is the potential that we may focus on studies that support our hypothesis (refactoring methods are unreliable) and ignore legitimate counterclaims. Therefore, we must analyze sources discussing refactoring and determine if they cast doubt on the reliability of refactoring methods.

While our initial analysis pointed toward refactoring methods being unreliable, that

does not mean all the commonly used methods cannot produce reliable results. For example, going by our current assumption that refactoring methods are being applied toward the incorrect domain, there could be domains where a refactoring method is consistently applied and produces results that align with the developer's expectations. Therefore, we must first gain an understanding of what areas the refactoring methods we demonstrate consistent results.

If the literature agrees with our initial hypothesis, we will have a stronger basis from which to base our study off of, given that there would be several different reports that refactoring reliability is in question. In addition, gathering these sources will help us identify common areas of misunderstanding amongst refactoring methods, which may be able to be identified in the investigation performed later in the study.

RQ2: What methods, languages, and IDEs are associated with "refactoring" posts?

For this question, we wish to see if the results from the literature review in RQ1 complement with information on how developers interpret different refactoring scenarios. This is done in order to verify if the refactoring methods discussed in the previous studies align with the interpretations developers have on refactoring in locations further removed from academia. The way this study chose to investigate these scenarios was to leverage Stack Overflow (https://stackoverflow.com), a Q&A that is a fairly popular location for developers to receive advice about many software topics, including refactoring. Developers will often come to the site to ask for advice on how to resolve bugs or improve their code, which are also common refactoring scenarios [1]. The scope of this question will be limited to situations in which developers specifically believe they are performing "refactoring" as that will allow us to observe what refactoring is conceptualized as by the developer and prevent us from injecting our own bias into the study by interpreting which actions constitute refactoring.

Here we chose to identify the refactoring methods, languages, and IDEs in posts that were specifically discussing "refactoring". For the refactoring methods, which were chosen from a well-known set of Fowler's refactoring methods [1], we want to see how often they

are mentioned within these posts to see if the common understanding of refactoring actions are aligned with actions discussed in real refactoring scenarios.

RQ3: Does the software domain affect which refactoring methods are applied?

Once we identify scenarios in which developers describe refactoring, we want to also take into consideration the domain of the questions. The domain, in this case, is the area of software development that the scenario covers, such as web design, OO-programming, or database modifications. We identify these domains by using LDA. LDA (Latent Dirichlet Allocation) is a topic modeling technique [8] that was used in several other studies [43, 34, 41, 5] in order to capture topics from Stack Overflow posts. By having these topics, we can show that there is a relationship between particular software areas and refactoring actions. If we can identify a trend between the actions and the domain, we may be able to more accurately recommend actions to refactor code.

For this question, we will identify the topics through performing LDA on the refactoring set of posts and then evaluate the topics discovered by the analysis. The topics will be manually identified by the author and a doctorate student with the rationale given in the following section for this question.

RQ4: What are the common characteristics of popular posts?

The final analysis we can perform is leveraging the popularity metric we recorded earlier to see what if there are any common trends amongst refactoring posts. The popularity metric represents a certain degree of user engagement, and we can assume that increased engagement represents an increased interest in the content in the post and an increased possibility that the post more accurately represents a form of ”refactoring”. We will be able to make this assumption in part because we will explicitly search for the ”refactor*” keyword in order to obtain the set of posts we perform the analysis on. The higher degree of engagement also suggests that that representation of refactoring may be closer to a consensus view of refactoring that is more easily understood by larger groups of developers. For that reason, we believe an investigation into the most popular posts is desirable in order to see there are any trends amongst their characteristics.

3.2 Literature Review

A literature review was performed to see how widespread the idea that refactoring methods can be ineffective is. This allows us to make sure that this study is not being unduly influenced by any specific group of studies as it is possible that the assumption that refactoring is ineffective does not have a wealth of evidence to back up the claim. Simultaneously, it will help us identify areas where refactoring has generally proven reliable or unreliable, which will help us see if those circumstances are present in Stack Overflow posts, enhancing our analysis of RQ2.

Since we initially believed the study done by Bavota et. al [2] was the earliest paper expressing doubts about the efficacy of refactoring, we wanted to observe if any other studies made similar claims. Sourcing papers that were related to this study alone would introduce bias into our sources, as it could be assumed that any paper citing it would likely have a similar view on refactoring or wished to use its results to back up its claims. Therefore, we also gathered sources that discussed refactoring by utilizing the Google Scholar database (https://scholar.google.com/). We initially searched for systematic literature reviews using the keywords ”refactor” and ”systematic” as those would provide a more holistic view of the state of refactoring at the time the study was performed and be in a position to notice trends (such as refactoring being ineffective) becoming more widespread.

Once these reviews were identified, we looked to see if they discussed any results from refactoring attempts in the study. We were looking for any mention of refactoring having been performed or a specific refactoring method being cited and the results of its use. We discarded any papers that did not contain a section discussing the results of refactoring or did not cite metrics or approaches on how the results came about.

3.3 Data Gathering

The data set used for this study came from the MSR 2019 Challenge Dataset accessed through SOTorrent[1], which is a set of Stack Overflow posts available to query through the use of Google's BigQuery platform [2]. In order to locate posts concerning refactoring, we attempted to search the body, title, and tags of each post for strings containing "refactor%". For question posts, we had the body, title, and tags were available, and for the answer posts, only the body of the post was available for parsing.

While we were aware of additional vocabulary related to refactoring (i.e., refactoring methods such as *Extract Method*, *Rename Method*, etc.), we searched specifically for "refactor%" in order to obtain the cases where the developers directly stated they were attempting some kind of refactoring and not changing the code for some other reason. Limiting the set in this way helped us gain an understanding of what actions developers believe refactoring consists of and what context encourages refactoring attempts. It also allows us to more reasonably assume that the posts are definitively talking about refactoring in some form, which helps us draw conclusions on the results to answer RQ2 and RQ4.

The answer posts were further divided between accepted answers and answers not accepted. In Stack Overflow, users have the option to mark an answer to a question that they ask as "accepted". We identify these accepted answers adding a join on our query to the Votes table, which records the date when an answer is accepted. Answers that were not accepted as answer posts that are related to a question about refactoring, but were not accepted by the question's user as an answer to their question.

3.4 Processing Post Data

There were three processing steps performed on Stack Overflow posts in this study: popularity calculation, parsing for relevant keywords, and LDA.

[1] https://empirical-software.engineering/projects/sotorrent/
[2] https://cloud.google.com/bigquery

3.4.1 Popularity

$$\mathbb{P} = \mathbb{S} + \mathbb{A} + \mathbb{C} + \mathbb{F} + \mathbb{V}$$

Figure 3.1: **Popularity Calculation for Stack Overflow Question posts from Pinto et. al 2015 study on energy consumption [31]**

The popularity calculation used in this study was taken from a study by Pinto et al. [31]. For this study, locating posts highly rated by this metric allows us to gain insight into what concepts of refactoring users engage the most with and allows us to answer RQ4 by manually reviewing highly popular posts. This calculation was preferred over similar calculations [42, 38] due to its inclusion of all the attributes of a post (i.e., score, comment count, etc.), which allowed the calculation to respond to all ways users might express interest in the post. The calculation is a simple aggregation of a post's score, answer count, favorite count, comment count, and view count. For answer posts where all these fields are not available, popularity can be calculated from the sum of the available fields: the score, answer count, and comment counts.

$$\mathbb{PA} = \mathbb{S} + \mathbb{C}$$

Figure 3.2: **Popularity Calculation for Stack Overflow Answer posts from Pinto et. al 2015 study on energy consumption [31]**

The studies using a popularity metric also suggested the need to normalize the composite values before calculating the result [31, 42, 38]. This is done to remove distortions in the result that are caused by the presence of particularly large outliers in the set [31]. For the same reason, we also pulled all the posts from the MSR 2019 Challenge dataset that we had gathered the refactoring questions from and obtained the mean of each of the values used in the popularity metric. The mean values were then compared against the mean values we would receive from the set of refactoring posts we identified.

Since the refactoring set was much smaller and had a greater number of outliers that could skew the results, we further processed the refactoring set to remove outliers using the outlierKD function (https://datascienceplus.com/identify-describe-plot-and-removing-the-outliers-from-the-dataset/) which was preferred due to ignoring the mean and standard deviation which would be overly influenced by posts with large amounts of engagement (i.e., score, comments, etc.). We did not perform this process on the full Stack Overflow set as we believed its size (45.8 million total posts) compared to the refactoring set we used for the study (101,163 total posts) allowed the mean values which would be used to be relatively unaffected.

Once the outliers were removed from the refactoring set and the means for all the values were obtained from the full Stack Overflow dataset, the refactoring set means for each composite value of the popularity metric were divided by the mean of the full dataset in order to see how the refactoring set differed from the average Stack Overflow post. The results of this process are discussed in the Experimental Results section of this paper.

Once the popularity was calculated for each type of post, we performed a separate manual review on the top 100 posts of each category. Since these posts demonstrate the highest level of engagement on the subject of ”refactoring”, it will give us insight into what developers consider related to refactoring. We also compared the least 20 posts each type as that can investigate why they may not have been as helpful or informative onto the concept of refactoring.

3.4.2 Parsing for Keywords

We further processed the posts in order to get relevant information on the refactoring actions taken and the context the actions were performed in. Understanding these will help us understand the context under which users talk about refactoring, which will help us answer RQ2. The information we were able to uncover in this manner was refactoring methods, programming languages, and IDEs.

Refactoring methods are standard methods used to describe the actions taken during

Table 3.1: Refactoring Methods per Language

Refactoring Method	**Word Stem**
Change Package	chang
Change Type	chang
Extract Class	extract
Extract Interface	extract
Extract Method	extract
Extract Superclass	extract
Extract Variable	extract
Inline Method	in line, in-line, inline, in-lined, inlined
Inline Variable	in line, in-line, inline, in-lined, inlined
Move Attribute	mov
Move Class	mov
Move Method	mov
Pull Up Attribute	pullup, pull up, pull-up, pulledup, pulled up, pulled-up
Pull Up Method	pullup, pull up, pull-up, pulledup, pulled up, pulled-up
Push Down Attribute	pushdown, push down, push-down, pusheddown, pushed down, pushed-down
Rename Class	rename
Rename Method	rename
Rename Package	rename
Rename Parameter	rename
Rename Variable	rename

refactoring. Some of the most well-known ones, such as *Extract Class* and *Rename Method* come from Fowler's original definitions of design patterns and code smells [1]. A list of common refactoring methods from these sources was used and is displayed in 3.1. These method names were reduced to their word stems to account for instances of different word forms (i.e., "changing", "renamed", "moving", etc.) and in the instance of "inline" refactoring, alternative spellings were included. This approach allowed us to capture all available instances where developers may have attempted to directly mention refactoring methods.

Since we were able to locate information in posts through keywords, we also attempted to look for a list of popular programming languages[3] and IDEs [4] as keywords. This will provide further avenues to analyze the posts and determine if refactoring is conceptualized differently amongst different languages or if it is influenced by the IDE at all. In any instances where there was a "/" character in the provided list (i.e., C/C++), we split the options into distinct languages. For instances with an "-" character, we added an additional permutation to our search in which it was replaced with a space character, which was the most common alternative we observed (i.e., Objective-C, Objective C). During the manual review, we also identified several tags and programming languages that were closely associated with the languages in the provided sources (i.e., "React" or "reactjs" for Javascript) and considered those languages equivalent instances.

The keywords were parsed by a small Java program that used regular expressions to find the keywords. The regex was in the form of ".*[method].*" for each refactoring method and "\b[language]\b" and "\b[IDE]\b" for each programming language and IDE respectively. The rationale was that refactoring methods have different permutations that need to be check (i.e., "move", "moved", "moving" for *Move Method*). Languages and IDEs were required to have defined word boundaries as there are popular programming languages that are spelled similarly (Java/Javascript, C/C#/C++), so a similar approach to the refactoring

[3] `http://pypl.github.io/PYPL.html`
[4] `http://pypl.github.io/IDE.html`

methods was not desirable. For the set of question posts, the keywords were searched for in the title, body, and tags of the post.

3.4.3 Topic Modeling

To derive and understand the topics of discussions that revolve around refactoring related posts, we performed topic modeling and n-gram analysis of the posts. Topic modeling is an unsupervised machine learning procedure that infers the topics (or thematic structure) discussed in large volumes of unlabeled and unstructured text documents [21]. N-grams are sets of co-occurring words (or letters), within a given window, that is available in a textual document and are useful in understanding a word in its context [22].

In order to perform the analysis described above, the posts were processed in order to format the text. This process involved the expansion of any word contractions (e.g., 'I'm' $\rightarrow$ 'I am') and the removal of URLs, code blocks, alphanumeric words, punctuation, and a list of stopwords supplied by NLTK [7]. We added some additional stopwords to our set to deal with common words in our dataset, with examples such as "thanks", "question", "answer". After this process, only the nouns, verbs, adjectives, and lemmatizations of potentially interesting words were retained. We opted to use lemmatization over stemming, as the lemma of a word is a valid English word [24] that we can interpret information from easier.

We then derived the topics discussed in each post by making use of the Latent Dirichlet Allocation (LDA) algorithm [8]. LDA has been used successfully in several studies involving Stack Overflow posts [43, 34, 41, 5] to perform topic modeling, so we felt comfortable using it in this study. LDA builds a statistical model that groups related words together from a corpus of textual documents where each grouping of frequently co-occurring words represents a topic. A mandatory input for the LDA algorithm is the number of topics that it is to generate. A low value will result in high level or general topics while a high value will produce more detailed topics, some of which will be noise. Hence, to arrive at the optimal number of topics, we iteratively extracted topics starting from two to fifty in increments of one. Each LDA execution cycle (i.e., model creation) was subjected to ten passes and one

hundred iterations.

Finally, to determine the optimal number of topics for our LDA analysis, we relied on a combination of topic coherence [33], perplexity [8], visualization [36], and manual analysis. With regards to our manual and visual analysis; we looked at the topics and terms generated in each execution cycle of the LDA algorithm to discover patterns in the topics such as similarities and overlapping of topics, topics that are consistent between each execution cycle, the prevalence of each topic, distribution and relevance of words by topics, etc. Finally, since the LDA process does not result in meaningful names for the topics it generates, we had to manually examine the list of generated terms to determine the appropriate topic names.

For our manual analysis, a collaborative approach was undertaken; we looked at the terms that represented each topic and came to an agreement on the name of the topics and also decided on the topics that were generated by noisy terms. We looked at the terms that are unique to each topic and the terms that are shared among topics (including the overall frequency of the term).

3.5 Manual Review

A manual review was performed on the posts obtained in order to accomplish two goals: (i) to identify any false positives from processing posts. (ii) within the post refactoring methods, and any languages or IDEs mentioned.

The languages and IDEs we looked for were taken from the same lists of popular programming languages and popular IDEs described above. Obtaining information on which of these languages or IDEs are more often involved in refactoring scenarios could help us identify scenarios that may better help developers understand the context in which they are refactoring and what alternative actions they may be able to take. If refactoring is more prevalent or rarer in certain domains than others, we also may be able to leverage the other information we have in the study to identify why that is so to make refactoring more efficient and reliable for other domains.

For this study, a sample set of 657 question posts was selected in order to give a confidence level of 99% and a confidence interval of 5 from a total set of 35,863 questions to perform the manual review. The results from this review will allow us to make an assumption on what developers were thinking about while refactoring and allow us to expand trends viewed in the sample onto the rest of the data, helping us answer RQ2 and RQ3. Additionally, a separate manual review will be performed on the top 100 popular question posts measured by the popularity metric described earlier.

Chapter 4

Experimental Results and Evaluation

4.1 RQ 1 Results

The earliest literature review that discussed quantitative impacts of refactoring methods was included in a thesis by Warenburg [44]. The literature review was conducted by gathering studies discussing code or design smells from Google Scholar, including the articles that discussed the empirical results of a refactoring method, discussed a method or tool that could be used for refactoring decisions or code smell analysis, or reported the usage of code smells in domains related to object-oriented software (i.e. UML diagram creation, architectural smells). In total, 46 studies were identified in this review.

At the time of writing, they could only find one study [35] citing several refactoring methods that were applied to an application and metrics were taken before or after to measure the improvement in maintainability. While changes and improvements, in metrics were observed, they were not linked directly to any maintainability concept (i.e. decrease in defect rates, decrease in time needed to add features to the product). The lack of this link means that a developer cannot easily determine what the impact of the refactoring on their project has for their overall work. Given the lack of studies at this time, we can assume that there was a general lack of understanding of what each refactoring method improved for the developer.

Al Dallal and Abdin performed a later study that attempted to synthesize the results of studies investigating how refactoring methods affected software quality attributes into a single literature review [3]. Including only studies that report empirically-based findings on

the impact of refactoring on software, they identified 76 primary studies. The studies were further divided based on several factors: i) whether they analyzed the impact of a single refactoring scenario or whether they investigated the impact of several refactoring actions together, ii) whether the study used internal quality attributes (cohesion, coupling) which can be measured by utilizing code artifacts or external quality attributes (maintainability, fault-proneness) which cannot be solely determined through code artifacts [14] to measure the impact of refactoring actions, iii) what approaches or statistical techniques were applied to measure the impact of refactoring on software quality, and iv) what datasets were used in the studies.

These divisions highlight the potential for misunderstanding on the impact of refactoring. Single refactoring actions are often ineffective in removing a code smell [9, 46], and in fact, have been reported to only partially remove the smell or even introduce new smells [9]. The division between internal and external quality attributes is notable as there are different set of measurable metrics that current ways of analyzing the code (i.e. static analysis tool) may be ineffective to effectively capture. Indeed, only 14 of the 76 studies even considered external quality attributes in their analysis.

Consistent definitions of metrics used for refactoring appear to be an issue in the current literature that hinder the applicability of the results to other studies. Only 10 studies were identified as applying some statistical techniques to analyze the impact of refactoring actions. This leads to the possibility that the majority of studies may focus on results that are more anecdotal or dependent on the understanding surrounding context of the software in order to effectively achieve the same results on separate projects. For datasets, the most notable result was the language distribution, with 87.2% of the 149 datasets used in all the studies were implemented in Java with the next most common languages being C++ (5) and C# (3). This leads us to assume that any refactoring methods may be more identifiable and effective within Java as there are the most examples to pull from.

Satnam and Paramvir performed an SLR on refactoring methods [23], focusing only on primary studies whose operations clearly targeted code smells and grouped the primary

studies observed based on whether statistical techniques were employed. It noticed that studies performing the same refactoring activity had varying perceived impacts, at times seeing a decrease or an improvement of metric values for the same action [12], depending on the quality measures used.

Interestingly, the impact of refactoring methods differed depending on whether studies focused on industrial or academic settings. Studies performed in academic settings typically found that their internal quality attributes improved after a refactoring activity was applied, while industrial settings saw no improvement in internal quality factors. These results are influenced by the small number of studies in industrial settings (17) compared to academic settings (125) as well as a tendency for the studies to focus on different datasets (industrial focusing on commercial datasets while academic studies usually prefer open-source projects), but that simply highlights that there appear to be different factors that determine the impact of refactoring in different contexts. Therefore, we can assume applying refactoring methods that produce positive results in academic contexts may fail to be as effective if they do not take into account their context and more influential measures that may impact the results.

This literature review proves that there has been consistent uncertainty within the software community about measuring the impact of refactoring methods. For the most part, this uncertainty is due to two contributing factors: i) lack of consistency in metrics, ii) the metrics being used not being extensive enough to cover all the scenarios and implications the metrics suggest. Our investigation into Stack Overflow posts should reinforce these findings, highlighting that there is often a lack of depth in the implications of particular refactoring actions.

4.2 RQ 2 Results

The manual review was performed over a representative sample set of 657 question posts that gave us a 99% confidence level with a 5% confidence interval. The discussion of the results will be split between the popularity metric, the languages identified, and the IDE.

Examples posts from Stack Overflow will be referenced in the following format (*Q*) to provide more context on the results and the interpretation. The number is the post id that can be searched on Stack Overflow to display the associated post.

4.2.1 Post Popularity

The results of the popularity calculation on the Stack Overflow and refactoring sets are displayed in Table 3. According to the results of our calculation, refactoring questions are 34.96% less popular than average Stack Overflow question posts, while accepted answers and answers that were not accepted were 31.72% and 29.11% as popular. Refactoring questions were viewed significantly less than Stack Overflow questions (received 85% fewer views) yet still maintained a fairly close answer count ratio of about 35% fewer answers on average. A less trafficked topic could be assumed to be harder to understand and provide a meaningful answer. This seems to imply a degree of familiarity with refactoring in the users that answer these questions as we would assume that there would be much fewer answers for a less well-known domain.

4.2.2 Refactoring Methods

Refer to 4 for the discussion of refactoring methods and 5 for discussion involving multiple refactoring methods mentioned in a post. The program we used initially received many false positives for the ”move” actions due to following the ”.*[method].*” regex pattern, mostly due to instances of ”remove” in the post. We were not able to identify any instances of the change, pull, or push methods in the question posts.

Extract methods were the most prevalent in the sample, with *Extract Superclass* (39), *Extract Class* (33), and *Extract Method* (26) being the most popular instance. This is in line with what other studies found [3], but it is not entirely clear what this means in the conception of refactoring. We can assume that in order to create effective change in the software, enacting one of these methods is necessary. Indeed, of the 26 refactoring actions that included more than one method, 23 include at least one Extract action.

Change (31) and Rename (19) were the next more present method identified in the sample. Interestingly, as opposed to the Extract refactors, they were mostly observed to be suggested independently. This makes sense as a renaming a class, for example, only involves renaming that specific class, and perhaps performing the same renaming operation to any references to it. We may be able to assume that if either of these categories is the suggested actions, the refactoring may be more narrowly defined and thus easier to interpret the consequences.

The Move (10) and Inline (16) methods were the next most present refactoring methods. Given their lack of entries in the sample, we may assume that these actions alone are inadequate for resolving refactoring issues discussed on Stack Overflow. While 4 out of the 10 cases of Move refactors did have at least one other refactor method applied, only 2 out of 10 applied for the Inline actions. It might be enlightening to follow up this investigation with evidence of what the developer intended to perform with this action to reveal why the use of these methods was not more common.

The Pull-Up and Push Down methods may be more prevalent than the results indicate due to the *Extract Superclass* and *Extract Class* method being so prevalent and having the potential to move attributes or methods along to the superclass or subclass once they are defined and extracted. This may imply that when developers mention refactoring methods, they may be considering several consecutive actions in order to achieve the desired effect. Since a lack of consideration of refactoring in batches is prevalent in the literature, it may be advisable for refactoring methods and tools to consider the impact of grouping of refactoring actions and present the actions as a sequence of events so that they are more aware of the phenomena.

Question posts often did not describe the refactoring methods that they were attempting to implement. In most cases, they accurately described the reason they were attempting to refactor the code (i.e., code optimization, implementing functionality, fixing a bug, etc.). However, they did not link it to one of the refactoring methods defined by Fowler [1]. This is in line with the results from other studies [9] and is also not unexpected due to those

definitions that may no longer be accurate to the current problems in software [38].

For the questions that did not have a specific instance of a refactoring method, there were three common patterns: coding advice, tool-based advice, and planned refactoring. Other instances where ”refactoring” was found in the posts were false positives where a refactoring keyword was present, but a refactor of the code was not intended (*Q16311092*).

Coding advice follows the scenario where the developer is attempting to accomplish a specific goal in the code, but does not describe the action needed to get there in the context of Fowler's refactoring methods. This is understandable in question posts as we could assume that a user who posts a question on Stack Overflow lacks information on what actions could resolve their issue. Developers typically had a clear idea of what they were trying to accomplish by refactoring, whether it was implementing some new functionality, resolve a bug they encountered, try to align the code with a design pattern or best practices.

Refactoring mentions were scenarios in which we found posts that mentioned the ”refactor*” keyword, but did not intend to actually perform any code change. Often this happened when users were attempting to figure out a best practice for a particular piece of code, which may result in changing the code, but there was no clear indication of what refactoring action they would take. Another common case is when ”refactor” was mentioned as a catch-all for a previous code change or an intention or plan to change the code or to remark that a section of code seemed in particular need of updating.

In these cases, the post was not attempting to address the code in question, but only to give context to a separate problem, such as an issue importing data or an inability to modify certain pieces of code to address the problem. The prevalence of these instances suggest that ”refactoring” is not interpreted as a strict formal definition and is instead used more colloquially to refer to changes in code.

4.2.3 Languages and IDEs

The results from the manual review of posts for languages and IDEs displayed in 6 for languages and 4.1 for IDEs. For languages, we have avoided displaying the number of false

positives for C#, R, and Go due to the program often misclassifying these instances. C# was not located by the program, but instead always classified as an instance of both C and C++. R was often misidentified due to looking for any instance of ”R” surrounded by non-alphanumeric characters, notable instances were regex patterns (*Q48619351*) or command line arguments (*Q48579214*). Go was also misidentified often as the word ”go” due to not accounting for cases in our regex pattern. Since we manually reviewed a representative sample set, we were still able to identify the true number of cases for these languages.

The most prevalent languages in the sample were Java (18.72%), C# (18.72%), JavaScript (17.05%), and Ruby (8.22%). Java, C#, and Ruby are OO-languages which maintain classes, interfaces, and other structures that are commonly mentioned in Fowler's refactoring methods [1]. JavaScript is a scripting language rather than OO-focused, but has a major implementation in AngularJS which makes use of inheritance for various structures such as controllers.

The most prevalent languages after the above were PHP, Python, SQL. Python was the most popular language listed in our source (http://pypl.github.io/PYPL.html), so it was initially expected that there would be a closer number of instances to languages such as Java and the rest above. More interesting still was that there was not a lot of refactoring methods recorded from the review. While Python does not have all the same structures as Java and C# (i.e., interfaces), it does have classes, variables, and methods. However, there was no particularly high distribution of these actions. This leads to the assumption that there is some difficulty in discussing refactoring within Python as since there the dataset gathered for the sample specifically looked for mentions of refactoring within the post. Given the lesser degree of refactoring instances observed, Fowler's set of refactoring methods may be poorly suited to dealing with changes that need to be made in Python systems.

One of the most interesting results we found was the prevalence of PHP in Stack Overflow questions. Although it was also ranked highly in our source list, it had as many entries as Python, which was the most popular language at the of writing. PHP is mostly web-focused but has class structures, and variables so it can enact refactoring along the lines of

our refactoring methods [1]. We may similarly assume that there is difficulty in discussing refactoring in PHP systems. Since PHP has a high association with web-based systems as well, it may be of interest to see if questions that discuss such scenarios do not have as many instances of refactoring that match to Fowler's methods.

SQL was not listed in the initial languages obtained for the set. However, it can be integrated with other languages that are making use of a database component, so its prevalence here is not particularly surprising. SQL instances were mostly discussing instances where developers were mostly attempting to improve the performance of queries or restructuring objects in the database. In instances where the posts were referring to proper instances of refactoring (i.e., intent clear and not asking about best practices/tools, *Q43393*), they often discussed issues that were analogous to scenarios in other languages such as attempting to implement new functionality (*Q6916646*) or move code off into a separate structure (i.e view) for a performance boost (*Q3598786*). These instances did not align with the refactoring methods that were used for the study and so none of the SQL posts were categorized as belonging to a refactoring method.

The IDEs did not seem to have a significant influence on refactoring methods. Visual Studio had the most instances in our sample with 30. IDEs typically implemented their own version of basic refactorings such as renaming a class in all locations in the program when a developer changes it. There are several instances we observed where developers were attempting to refactor through IDEs (*Q12579561*, *Q40715010*, *Q14343054*) but encountered difficulties, so it appears that developers do tend to think refactoring through these tools is a topic of interest.

4.3 RQ3 Results

The LDA analysis identified 4 topics: Object-Oriented, Environment, Database, and Front-End. Object-Oriented represents common coding scenarios that involve the creation and modification of code elements. Common terms include "array", "value", or "function".

Table 4.1: IDEs in Sample Set

IDE Name	Count
Android Studio	5
Eclipse	19
Emacs	2
IntelliJ	7
MonoDevelop	1
PhpStorm	1
PyCharm	2
Vim	2
Visual Studio	30
Visual Studio Code	2
Xcode	11

Table 4.2: Refactoring Methods per Topic

Topic \Refactoring Methods	**Change Package**	**Change Type**	**Extract Class**	**Extract Interface**	**Extract Method**	**Extract Superclass**	**Extract Variable**	**Inline Method**	**Inline Variable**	**Move Attribute**	**Move Class**	**Move Method**	**Pull Up Attribute**	**Pull Up Method**	**Push Down Attribute**	**Push Down Method**	**Rename Class**	**Rename Method**	**Rename Package**	**Rename Parameter**	**Rename Variable**	**Total**
Database (143)	0	6	1	1	2	2	0	3	1	0	0	1	0	0	0	0	0	0	0	0	0	17
Environment (164)	5	1	8	1	3	1	0	1	0	0	1	0	0	0	0	0	4	1	4	0	3	33
Front-End (75)	0	0	1	1	1	2	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	7
Object-Oriented (275)	4	16	23	14	20	14	2	6	5	2	0	5	0	1	1	0	2	2	0	1	2	120

Database involves scenarios with data components such as queries, databases, and connections to the database. These scenarios typically do not have associated refactoring methods associated with them due to the difference between common programming structures (i.e., class, variable, method) and database structures (i.e., tables, queries). Web Development involves scenarios that are specifically focused on developing and addressing issues within webpages, such as dealing with views, website design, and media content (i.e., images, videos).

Object-Oriented made up the majority of topics in the dataset (41.86%) and also had the most amount of identified refactoring methods. Since the topic has the most relation to coding objects such as arrays, strings, etc. it makes sense for this to have the most recognized refactoring methods. Similarly to the languages, it seems the Extract refactorings are the most common with 73 entries, reinforcing that these refactoring seem particularly useful for explaining the necessary code changes to cause the desired impact.

Environment (24.96%) appears to have the most relation to tools and development environments in the system. Often, the questions within this topic concern attempting to achieve refactoring through the use of these tools (*Q16883286*, *Q35100985*, *Q52992990*). Since most IDEs implement some form of refactoring functionality, the prevalence of this topic makes sense.

Database (21.77%) had the second-lowest count of recorded refactoring methods at 18 and represented 21.77% of the sample set. This makes sense as database components are not as applicable to Fowler's refactoring methods due to not containing the same structures.

Front-End was the least prevalent area in our sample, representing only 11.41% of the sample. There were only 5 instances of verified refactoring methods that fit into Fowler's categories in for this topic. Since the structures described in the methods (e.g., class, instances, package) are not commonly used within web development, the lack of results for this topic makes sense.

The interesting interpretation of these results is that the topic seems to have a large

Table 4.3: Top 100 Popular Questions: Refactoring Methods

Popular Questions	Change Package	Change Type	Extract Class	Extract Interface	Extract Method	Extract Superclass	Extract Variable	Inline Method	Inline Variable	Move Attribute	Move Class	Move Method	Pull Up Attribute	Pull Up Method	Push Down Attribute	Push Down Method	Rename Class	Rename Method	Rename Package	Rename Parameter	Rename Variable	Total
-	0	5	3	0	9	2	3	2	2	0	0	1	0	0	0	1	0	1	1	0	1	31

influence on how relevant the refactoring methods are to it. 120 of the 177 identified refactoring methods existed in the Object-Oriented topic, which most exhibited common coding structures (i.e., classes, interfaces, etc.). All other topics saw a sharp drop in the number of recognized refactorings suggesting that Fowler's methods might not be able to accurately encompass the changes desired in these domains. This highlights the need for analysis and suggestion of refactoring methods to be more aware of the context to which they are applied in order to ensure that their application makes sense.

4.4 RQ4 Results

Within this set of popular posts, 31 refactoring methods were identified. There were only 5 instances of multiple refactoring methods observed. 22 of the questions were identified as false positives and the remaining 47 posts did not have an identifiable refactoring method. The most common trait found in the false positives was a difficulty in managing a tool or framework and using "refactor" to denote the attempt to troubleshoot the problem (e.g. *Q7887580*, *Q1946364*, *Q20689979*). Other common cases were attempts by developers to identify a best practice, but without the context of any code that they wish to apply it to (e.g. *Q15300521*, *Q15260774*). For the identified refactoring methods, Extract refactors appear to be the most dominant as well (17) followed by Change (5) and Inline (4).

Interestingly, the most popular posts did not show a large trend of having posts with the

Table 4.4: Top 100 Popular Questions: Refactoring Topics

Topics\Ref. Methods	Change Package	Change Type	Extract Class	Extract Interface	Extract Method	Extract Superclass	Extract Variable	Inline Method	Inline Variable	Move Attribute	Move Class	Move Method	Pull Up Attribute	Pull Up Method	Push Down Attribute	Push Down Method	Rename Class	Rename Method	Rename Package	Rename Parameter	Rename Variable	Total
Database (13)	0	2	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	3
Environment (28)	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	1	4
Front-End (7)	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2
Object-Oriented (52)	0	3	3	0	7	2	2	2	2	0	0	1	0	0	0	1	0	0	0	0	0	23

most words or the most code snippets as being the most popular, against the trend observed by Nasehi et. al in their previous study [28]. Only 9 posts were observed with 3 or more code blocks. If we assume the posts that are the most popular are representative of an ideal way to communicate a refactoring issue, we can interpret from this that a post does not need to have a great deal of code detail in order to appear interesting to developers.

In cases where the refactoring method was not identified, the most common theme amongst the posts was an attempt to identify a best practice for their code. Performance issues often stuck out as the most common reason to attempt a refactor, citing concerns or uncertainty if a particular piece of code would cause issues if left as is (*Q35959259*).

The topic distribution on the most popular questions is similar to distribution in the sample set with Object-Oriented (52) representing the overwhelming majority of topics followed by Environment (28), Database (13), and Front-End (7). 21 false positives were identified, with the majority coming from the Object-Oriented (11) and the Environment (9) topics. The remaining two topics had a single false positive each.

The distribution is similar to the sample set (popularity: 13%, 28%, 7%, 52% — sample set: 21.77%, 24.96%, 11.41%, 41.86%). Most of the difference can be attributed to the popularity set not being a representative sample of the posts, but the proportions are not too far off from the sample's.

Chapter 5

Threats to Validity and Future Work

5.1 Threats to Validity

The analysis of posts was not extended beyond popularity for the answer posts in this study. Since question posts are often more vague and unsure of what they want to accomplish and the techniques (i.e. refactoring methods) due to trying to identify and explain a problem for a broader audience, answer posts may have more direct insight into what actions should be pursued in order to address the issue. Due to the time it took to conduct the analysis of the results for the question posts, we were reduced to only including the question results at this time. we acknowledge that the analysis of question posts alone does not give us a full pictures of how developers' discuss refactoring.

Since we have also acquired the answer posts associated with refactoring, we are able to conduct a future study to investigate them. The results from the questions that we have analyzed here will be able to be supplemented by later studies.

The set of refactoring methods used in the study is taken from the list of methods introduced by Fowler [1]. There have since been several additional methods identified since their introduction, and it is possible that we may fail to properly catalog every instance of refactoring we come across. However, for the purposes of this study, we believe that the set we have chosen is sufficiently broad to cover most forms of refactoring we come across. Searching for the word stems of these methods also allowed us to pick up on any permutations of the method that might have been outside of the set we investigated. While we may miss refactors tailored to more specific domains (i.e. databases), the lack of results

in our studies will highlight which areas may need different sets of methods to properly describe their refactoring actions.

As a result of our study, we identified that ”refactoring” was used in a more colloquial way to refer to any changes in the code, rather than a strict formal definition. This damages any assertions that refactoring is considered to be attached to any particular refactoring methods such as Fowler’s [1] and suggests that any vocabulary describing a change in this setting may better locate instances of refactoring rather than looking strictly for mentions of ”refactor*”. While ”refactor” still suggests some sort of intent to change the code as reinforced by our manual review, we did not verify if there is was any other vocabulary that is more accurate or inclusive than ”refactor”.

As part of analysis of the posts, we were able to obtain data about the word frequency in regards to identifying the topics in the posts. We may be able to leverage this to identify words that reliably correlate with direct mentions of refactoring to create a new set of posts to investigate.

The popularity calculation we used is borrowed from the work done by Pinto et. al [31] and is a simple aggregation of several post values. There is a possibility among these values is particularly large and overly influences the other popularity calculations. We performed normalization over each of the values to mitigate this risk, but there is still the possibility that a particularly large outlier (i.e. a highly scored and viewed post) could unfairly influence the popularity result. By recording all of these values however, we were able to observe instances where that had occurred and can address those in future studies.

There is also the issue that some of the values used for the may not be as relevant as others. Although they all represent a degree of user engagement and interest, there is no guarantee that these particular values are actually influential. There might also be areas that we had not recorded which may be influential (i.e. length of post, time of posting). Again, recording all the information here will at least allow us to recognize if there is some sort of discrepency between their influence for future study.

When processing the post data for popularity, we did not remove the outliers of the

Stack Overflow set as we had done similarly for the set of refactoring posts. While we realized that this could skew the results, the sheer size of MSR 2019 Challenge database (45.8 million total posts) compared to the refactoring set we used for the study (101,163 total posts) assured us that any disturbance by the outliers would be mitigated.

The program we had built to identify the programming languages present in the post had issues with detecting instances of C# and Go, misidentifying them as instances of C and C++ together and as the word ”go” respectively. Similarly, the ”move” method cases in our sample set were often misidentified from common words such as ”remove”. Since we also performed manual review on a representative sample with a 99% confidence level, we were still able to identify these false positives and make assumptions about the rest of the set. We will not be able to completely ensure that these languages are not more prevalent in the rest of the set by the program results also because of these results, but we are still able to make assumptions from the rest of the languages which did not display this issues.

5.2 Future Work

The Stack Overflow analysis from this study is part of a further study analyzing the content of posts in order to discern what developers are thinking about while attempting to perform refactoring. The study is currently planned to replicate the post analysis displayed in this thesis onto answer posts, duplicating the processing to reveal topics and a manual review of a representative sample in order to discover the languages and refactoring methods that are suggested in these scenarios.

The study also plans to apply maintenance categories [25] and evolution tasks [18] alongside the previous analysis. Their addition will help us further interpret the developers' goals for attempting refactoring and will also be verified through manual review. This will give us more context as to what the developers are trying to accomplish in each instance, giving us a further means to identify how the refactoring methods can be more properly applied.

Finally, since the number of posts selected to manually analyze for popularity was

arbitrary, we wish to expand the number of posts investigated to 1% of the total refactoring question set. We also plan to replicate this analysis with the answer posts within this investigation as well.

Chapter 6

Conclusion

Our investigation into the prevalence of the ineffectiveness of refactoring as a topic revealed that there were several issues that were commonly attested to contribute to the fuzziness of the results of refactoring actions. First, the observation of refactoring actions usually occur in isolation. Observing the refactoring methods in the set showed that the Extract actions (i.e. *Extract Class*, *Extract Interface*, *Extract Superclass*) are considered a common way to achieve some change in the code, but only 30.26% of Extract refactorings (23 out of 76) were taken alongside other actions. This implies that refactoring actions as they are commonly defined are assumed to consist of a series of actions to achieve a desired effect (i.e. removal of a code smells, conforming to a best practice).

While the resolution of potentially complex problems seems like it would take multiple actions to resolve, code refactorings are often not discussed as grouping of actions and instead considered individually. In order to increase reliability of these methods, an identification of whether methods are being individually or if they are part of a set of actions must be recognized in order to make sure developers have the correct assumption on what should occur during refactoring.

Second, the topic seems to have a large influence in how relevant the refactoring methods are to it. 120 of the 177 identified refactoring methods existed in the Object-Oriented topic, which most exhibited common coding structures (i.e. classes, interfaces, etc.). All other topics saw a sharp drop in the amount of recognized refactorings suggesting that Fowler's methods might not be able to accurately encompass the changes desired in these domains. This highlights the need for analysis and suggestion of refactoring methods to be

more aware of the context to which they are applied in order to ensure that their application makes sense.

Of particular interest is the expansion or utilization of refactoring methods that take into the context of web development. Front-end focused questions had the least amount of methods that could be identified by them, even if the languages contained structures that aligned with refactoring set that we were using (classes, variables) such as in PHP. It might be imperative to see if there are any candidate refactoring sets that capture the actions developers want to perform.

This study highlights the need to take multiple conceptualizations of refactoring into consideration when considering their effectiveness. In particular, increasing the awareness that batch refactoring can have potential advantages over a single refactoring, evaluating the different metrics used to determine software quality to see if they are applicable in their particular domain, and taking the user perception of the issue within the software to help guide them toward a better understanding of the issue and what actions would be effective in addressing it.

Bibliography

[1] *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., USA, 1999.

[2] An experimental investigation on the innate relationship between quality and refactoring. *J. Syst. Softw.*, 107(C):1–14, September 2015.

[3] J. Al Dallal and A. Abdin. Empirical evaluation of the impact of object-oriented code refactoring on quality attributes: A systematic literature review. *IEEE Transactions on Software Engineering*, 44(1):44–69, 2018.

[4] Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, Ali Ouni, and Marouane Kessentini. Do design metrics capture developers perception of quality? an empirical study on self-affirmed refactoring activities. *ArXiv*, abs/1907.04797, 2019.

[5] M. Alshangiti, H. Sapkota, P. K. Murukannaiah, X. Liu, and Q. Yu. Why is developing machine learning applications challenging? a study on stack overflow posts. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–11, 2019.

[6] Sven Amani, Sarah Nadi, Hoan A. Nguyen, Tien N. Nguyen, and Mira Mezini. Mubench: a benchmark for api-misuse detectors. In *Proceedings of the 13th International Workshop on Mining Software Repositories - MSR '16*, pages 464–467, 2016.

[7] Steven Bird. Nltk: The natural language toolkit. *ArXiv*, cs.CL/0205028, 2002.

[8] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. Latent dirichlet allocation. *J. Mach. Learn. Res.*, 3(null):993–1022, March 2003.

[9] Diego Cedrim, Alessandro Garcia, Melina Mongiovi, Rohit Gheyi, Leonardo Sousa, Rafael de Mello, Baldoino Fonseca, Márcio Ribeiro, and Alexander Chávez. Understanding the impact of refactoring on smells: A longitudinal study of 23 software projects. page 465–475, 2017.

[10] Diego Cedrim, Leonardo Sousa, Alessandro Garcia, and Rohit Gheyi. Does refactoring improve software structural quality? a longitudinal study of 25 projects. In *Proceedings of the 30th Brazilian Symposium on Software Engineering*, SBES '16, page 73–82, New York, NY, USA, 2016. Association for Computing Machinery.

[11] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[12] Mel Ó Cinnéide, Iman Hemati Moghadam, Mark Harman, Steve Counsell, and Laurence Tratt. An experimental search-based approach to cohesion metric evaluation. *Empirical Software Engineering*, 22:292–329, 2016.

[13] A. Eposhi, W. Oizumi, A. Garcia, L. Sousa, R. Oliveira, and A. Oliveira. Removal of design problems through refactorings: Are we looking at the right symptoms? In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pages 148–153, May 2019.

[14] Norman Fenton and Shari Lawrence Pfleeger. *Software Metrics (2nd Ed.): A Rigorous and Practical Approach*. PWS Publishing Co., USA, 1997.

[15] Manuele Ferreira, Eiji Barbosa, Isela Macia, Roberta Arcoverde, and Alessandro Garcia. Detecting architecturally-relevant code anomalies: A case study of effectiveness and effort. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, SAC '14, page 1158–1163, New York, NY, USA, 2014. Association for Computing Machinery.

[16] F. A. Fontana, J. Dietrich, B. Walter, A. Yamashita, and M. Zanoni. Antipattern and code smell false positives: Preliminary conceptualization and classification. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 609–613, March 2016.

[17] Rachel Harrison, Steve J. Counsell, and Reuben V. Nithi. An evaluation of the mood set of object-oriented software metrics. *IEEE Trans. Softw. Eng.*, 24(6):491–496, June 1998.

[18] L. P. Hattori and M. Lanza. On the nature of commits. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering - Workshops*, pages 63–71, 2008.

[19] M. Hozano, A. Garcia, N. Antunes, B. Fonseca, and E. Costa. Smells are sensitive to developers! on the efficiency of (un)guided customized detection. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, pages 110–120, 2017.

[20] N. Imtiaz, A. Rahman, E. Farhana, and L. Williams. Challenges with responding to static analysis tool alerts. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 245–249, 2019.

[21] B. Johnston, A. Jones, and C. Kruger. *Applied Unsupervised Learning with Python: Discover hidden patterns and relationships in unstructured data with Python*. Packt Publishing, 2019.

[22] D. Jurafsky and J.H. Martin. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. Prentice Hall series in artificial intelligence. Pearson Prentice Hall, 2009.

[23] Satnam Kaur and Paramvir Singh. How does object-oriented code refactoring influence software quality? research landscape and challenges. *Journal of Systems and Software*, 157:110394, 08 2019.

[24] H. Lane, H. Hapke, and C. Howard. *Natural Language Processing in Action: Understanding, Analyzing, and Generating Text with Python*. Manning Publications Company, 2019.

[25] B. P. Lientz, E. B. Swanson, and G. E. Tompkins. Characteristics of application software maintenance. *Commun. ACM*, 21(6):466–471, June 1978.

[26] Isela Macia, Roberta Arcoverde, Alessandro Garcia, Christina Chavez, and Arndt von Staa. On the relevance of code anomalies for identifying architecture degradation symptoms. In *Proceedings of the 2012 16th European Conference on Software Maintenance and Reengineering*, CSMR '12, page 277–286, USA, 2012. IEEE Computer Society.

[27] E. Murphy-Hill, C. Parnin, and A. P. Black. How we refactor, and how we know it. *IEEE Transactions on Software Engineering*, 38(1):5–18, Jan 2012.

[28] S. M. Nasehi, J. Sillito, F. Maurer, and C. Burns. What makes a good code example?: A study of programming q a in stackoverflow. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 25–34, 2012.

[29] F. Palomba, G. Bavota, M. D. Penta, R. Oliveto, and A. D. Lucia. Do they really smell bad? a study on developers' perception of bad code smells. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 101–110, 2014.

[30] Jevgenija Pantiuchina, Michele Lanza, and Gabriele Bavota. Improving code: The (mis) perception of quality metrics. *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 80–91, 2018.

[31] Gustavo Pinto, Fernando Castor, and Yu David Liu. Mining questions about software energy consumption. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, page 22–31, New York, NY, USA, 2014. Association for Computing Machinery.

[32] Kyle Prete, Napol Rachatasumrit, Nikita Sudan, and Miryung Kim. Template-based reconstruction of complex refactorings. In *Proceedings of the 2010 IEEE International Conference on Software Maintenance*, ICSM '10, page 1–10, USA, 2010. IEEE Computer Society.

[33] Michael Röder, Andreas Both, and Alexander Hinneburg. Exploring the space of topic coherence measures. In *Proceedings of the Eighth ACM International Conference on Web Search and Data Mining*, WSDM '15, pages 399–408, New York, NY, USA, 2015. ACM.

[34] Christoffer Rosen and Emad Shihab. What are mobile developers asking about? a large scale study using stack overflow. *Empirical Softw. Engg.*, 21(3):1192–1223, June 2016.

[35] S. V. Shrivastava and V. Shrivastava. Impact of metrics based refactoring on the software quality: a case study. In *TENCON 2008 - 2008 IEEE Region 10 Conference*, pages 1–6, 2008.

[36] Carson Sievert and Kenneth Shirley. Ldavis: A method for visualizing and interpreting topics. In *Proceedings of the workshop on interactive language learning, visualization, and interfaces*, pages 63–70, 2014.

[37] Danilo Silva, Nikolaos Tsantalis, and Marco Tulio Valente. Why we refactor? confessions of github contributors. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, page 858–870, New York, NY, USA, 2016. Association for Computing Machinery.

[38] Amjed Tahir, Aiko Yamashita, Sherlock Licorish, Jens Dietrich, and Steve Counsell. Can you tell me if it smells? a study on how developers discuss code smells and anti-patterns in stack overflow. In *Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering 2018*, EASE'18, page 68–78, New York, NY, USA, 2018. Association for Computing Machinery.

[39] Nikolaos Tsantalis. Refactoringminer github page, 2017.

[40] Nikolaos Tsantalis, Victor Guana, Eleni Stroulia, and Abram Hindle. A multidimensional empirical study on refactoring activity. In *Proceedings of the 2013 Conference of the Center for Advanced Studies on Collaborative Research*, CASCON '13, page 132–146, USA, 2013. IBM Corp.

[41] Isabel K. Villanes, Silvia M. Ascate, Josias Gomes, and Arilo Claudio Dias-Neto. What are software engineers asking about android testing on stack overflow? In *Proceedings of the 31st Brazilian Symposium on Software Engineering*, SBES'17, page 104–113, New York, NY, USA, 2017. Association for Computing Machinery.

[42] Shaohua Wang, Iman Keivanloo, and Ying Zou. How do developers react to restful api evolution? In Xavier Franch, Aditya K. Ghose, Grace A. Lewis, and Sami Bhiri, editors, *Service-Oriented Computing*, pages 245–259, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[43] Shaowei Wang, David Lo, and Lingxiao Jiang. An empirical study on developer interactions in stackoverflow. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC '13, page 1019–1024, New York, NY, USA, 2013. Association for Computing Machinery.

[44] Ruben Wangberg. A literature review on code smells and refactoring, 2010.

[45] A. Yamashita and L. Moonen. Do developers care about code smells? an exploratory survey. In *2013 20th Working Conference on Reverse Engineering (WCRE)*, pages 242–251, 2013.

[46] Aiko Yamashita and Leon Moonen. Exploring the impact of inter-smell relations on software maintainability: An empirical study. pages 682–691, 05 2013.

[47] Vanius Zapalowski, Daltro José Nunes, and Ingrid Nunes. Understanding architecture non-conformance: Why is there a gap between conceptual architectural rules and

source code dependencies? In *Proceedings of the XXXII Brazilian Symposium on Software Engineering*, SBES '18, page 22–31, New York, NY, USA, 2018. Association for Computing Machinery.

[48] Tianyi Zhang, Ganesha Upadhyaya, Anastasia Reinhardt, Hridesh Rajan, and Miryung Kim. Are code examples on an online qa forum reliable? a study of api misuse on stack overflow. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, page 886–896, New York, NY, USA, 2018. Association for Computing Machinery.

Appendix A

Table 1: Refactoring SLRs

Title	Year	Accessible	Fits Criteria
An empirical study of the bad smells and class error probability in the post-release object-oriented system, evolution Wei Li and Raed Shatnawi	2007	Y	N
A Literature Review on Code Smells and Refactoring Ruben Wangberg	2010	Y	Y
Identifying the move method refactoring opportunities based on evolutionary algorithm Wei-Feng Pan, Jing Wang, Mu-Chou Wang	2013	N	-
Recommending Move Method Refactorings Using Dependency Sets Vitor Sales, Ricardo Terra, Luis Fernando Miranda, Marco Tulio Valente	2013	Y	N
Software fault prediction metrics: A systematic literature review Danijel Radjenović, Marjan Heričko, Richard Torkar, Aleš Živkovič	2013	N	-
Trends, opportunities and challenges of software refactoring: A systematic literature review Mesfin Abebe and Cheol-Jung Yoo	2014	N	-
Identifying refactoring opportunities in object-oriented code: A systematic literature review Jehad Al Dallal	2014	-	-
A review of code smell mining techniques Ghulam Rasool and Zeeshan Arshad	2015	Y	N
Non-Source Code Refactoring: A Systematic Literature Review Siti Rochimah, Siska Arifiani, Vika Insanittaqwa	2015	Y	N
A systematic literature review: Refactoring for disclosing code smells in object oriented software Satwinder Singh and Sharanpreet Kaur	2017	Y	N
Empirical Evaluation of the Impact of Object-Oriented Code Refactoring on Quality Attributes: A Systematic Literature Review Jehad Al Dallal and Anas Abdin	2017	Y	Y
A systematic literature review on the detection of smells and their evolution in object-oriented and service-oriented systems Fatima Sabir, Francis Palma, Ghulam Rasool, Yann-Gaël Guéhéneuc, Naouel Moha	2019	Y	N
A Systematic Literature Review on Empirical Analysis of the Relationship Between Code Smells and Software Quality Attributes Amandeep Kaur	2019	N	-
Categorization Refactoring Techniques based on their Effect on Software Quality Attributes Abdullah Almogahed, Mazni Omar, Nur Zakaria	2019	Y	N
How does Object-Oriented Code Refactoring Influence Software Quality? Research Landscape and Challenges Satnam Kaur and Paramvir Singh	2019	Y	Y

Table 2: Topics: Set of Terms

Topic	List of Words
Database	database, table, query, sql, db, json
Environment	package, studio, eclipse, directory, tool, maven
Front-End	javascript, event, button, html, thread, callback, promise
Object-Oriented	class, interface, constructor, method, function, object, generic

Table 3: Statistical summary of popularity characteristics of question and answer posts

Measurement Name	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
All Stack Overflow Questions (total instances: 18,154,493)						
Question Score	-1	0	0	2.04	1	23361
Answer Count	0	1	1	1.52	2	518
Comment Count	0	0	1	2.00	3	109
Favorite Count	0	0	0	0.61	0	10635
View Count	2	77	286	2282.49	1010	8083391
Popularity Score	-37.22	1.33	2.48	5.00	4.34	29359.10
Refactoring Questions (total instances: 35,863)						
Question Score	-3	0	1	1.04	2	5
Answer Count	0	1	1	1.33	2	3
Comment Count	0	0	1	1.66	3	7
Favorite Count	0	0	0	0.25	0	2
View Count	3	68	171	356.12	486	1897
Popularity Score	-2.71	1.46	2.66	3.25	4.45	10.86
All Stack Overflow Accepted Answers (total instances: 9,512,864)						
Accepted Answer Score	-129	1	1	4.60	3	30552
Comment Count	0	0	1	1.94721	3	157
Popularity Score	-24.47	0.43	1.03	2.00	2.19	6687.15
Refactoring Accepted Answers (total instances: 21,781)						
Accepted Answer Score	-3	1	2	2.11	3	8
Comment Count	0	0	1	1.59	2	7
Popularity Score	-0.44	0.44	1.09	1.37	1.98	4.90
All Stack Overflow Non-Accepted Answers (total instances: 18,152,145)						
Non-Accepted Answer Score	-58	0	0	1.94	2	10369
Comment Count	0	0	0	1.13	2	118
Popularity Score	-10.56	0	0.44	1.00	1.24	2273.57
Refactoring Non-Accepted Answers (total instances: 43,969)						
Non-Accepted Answer Score	-3	0	1	1.02	2	5
Comment Count	0	0	0	0.88	1	5
Popularity Score	-1.74	0	0.44	0.71	1.09	3.26

Table 4: Refactoring Method

Refactoring Method	**Count**
Change	31
- Change Package	9
- Change Type	23
Extract	76
- Extract Class	33
- Extract Interface	17
- Extract Method	26
- Extract Superclass	19
- Extract Variable	2
Inline	16
- Inline Method	10
- Inline Variable	6
Move	10
- Move Attribute	3
- Move Class	1
- Move Method	6
Pull Up	1
- Pull Up Attribute	0
- Pull Up Method	1
Push Down	1
- Push Down Attribute	1
- Push Down Method	0
Rename	19
- Rename Class	7
- Rename Method	3
- Rename Package	4
- Rename Parameter	1
- Rename Variable	5

Table 5: Cases of Multiple Refactoring Methods in Stack Overflow Posts

Refactoring Method	**Count**
Change	2
- Change Package, Change Type	1
- Change Package, Extract Interface, Extract Superclass	1
Extract	23
- Change Package, Extract Interface, Extract Superclass	1
- Extract Class, Extract Interface, Extract Superclass	2
- Extract Class, Extract Interface, Move Attribute	1
- Extract Class, Extract Method	5
- Extract Class, Extract Method, Extract Superclass	1
- Extract Class, Extract Superclass	2
- Extract Class, Pull Up Method	1
- Extract Interface, Extract Superclass	6
- Extract Interface, Move Method	2
- Extract Method, Inline Method	1
- Extract Method, Inline Variable	1
Inline	2
- Extract Method, Inline Method	1
- Extract Method, Inline Variable	1
Move	4
- Extract Class, Extract Interface, Move Attribute	1
- Extract Interface, Move Method	2
- Move Class, Rename Class	1
Pull Up	1
- Extract Class, Pull Up Method	1
Push Down	0
Rename	2
- Move Class, Rename Class	1
- Rename Class, Rename Method	1

Table 6: Languages in Sample Set

Language	Count	False Positive
ASP.NET	9	0
C	15	0
C#	123	0
C++	25	0
CSS	8	0
Dart	1	0
Delphi	3	0
Erlang	1	0
F#	1	0
Fortran	1	0
Go	5	0
HTML	11	0
Java	123	0
JavaScritpt	112	0
Kotlin	3	0
Lua	1	0
Objective-C	14	0
PHP	33	0
Perl	4	0
Powershell	3	0
Python	31	0
R	5	0
Ruby	54	0
Rust	3	0
Scala	9	0
Shell	1	0
SQL	16	0
Swift	9	0
TypeScript	2	0
Visual Basic	4	0

Table 7: Refactoring Methods per Language

Language	Change Package	Change Type	Extract Class	Extract Interface	Extract Method	Extract Superclass	Extract Variable	Inline Method	Inline Variable	Move Attribute	Move Class	Move Method	Pull Up Attribute	Pull Up Method	Push Down Attribute	Push Down Method	Rename Class	Rename Method	Rename Package	Rename Parameter	Rename Variable	Total
ASP.NET	0	2	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4
C	0	0	0	0	1	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	3
C#	2	6	8	9	3	7	1	1	1	1	0	2	0	1	0	0	1	1	0	0	0	44
C++	0	0	2	1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	5
CSS	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
Dart	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Delphi	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1
Erlang	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
F#	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
Fortran	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Go	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
HTML	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Java	3	8	7	3	7	4	0	3	2	1	0	0	0	0	0	0	3	0	1	0	2	44
JavaScritpt	1	3	6	0	2	2	0	2	0	0	0	0	0	0	0	0	0	0	0	0	1	17
Kotlin	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	2
Lua	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Objective-C	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	2
PHP	1	2	3	1	1	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	11
Perl	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	2
Powershell	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Python	1	1	1	0	3	0	0	0	0	0	0	2	0	0	1	0	0	0	0	0	0	9
R	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Ruby	0	0	1	1	3	0	0	1	2	0	0	1	0	0	0	0	1	1	0	0	0	11
Rust	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Scala	0	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3
Shell	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
SQL	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
Swift	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2
TypeScript	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Visual Basic	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1