Noname manuscript No.
(will be inserted by the editor)

How Do I Refactor This? An Empirical Study on Refactoring Trends and Topics in Stack Overflow

Anthony Peruma · Steven Simmons · Eman Abdullah AlOmar · Christian D. Newman · Mohamed Wiem Mkaouer · Ali Ouni

Received: date / Accepted: date

Abstract An essential part of software maintenance and evolution, refactoring is performed by developers, regardless of technology or domain, to improve the internal quality of the system, and reduce its technical debt. However, choosing the appropriate refactoring strategy is not always straightforward, resulting in developers seeking assistance. Although research in refactoring is well-established, with several studies altering between the detection of refactoring opportunities and the recommendation of appropriate code changes, little is known about their adoption in practice. Analyzing the perception of developers is critical to understand better what developers consider to be problematic in their code and how they handle it. Additionally, there is a need for bridging the gap between refactoring, as research, and its adoption in practice, by extracting common refactoring intents that are more suitable for what developers face in reality. In this study, we analyze refactoring discussions on Stack Overflow through a series of quantitative and qualitative experiments. Our results show that Stack Overflow is utilized by a diverse set of developers for refactoring assistance for a variety of technologies. Our observations show five areas that developers typically require help with refactoring– Code Optimization, Tools and IDEs, Architecture and Design Patterns, Unit Testing, and Database. We envision our findings better bridge the support between traditional (or academic) aspects of refactoring and their real-world applicability, including better tool support.

Keywords Empirical Study · Software Maintenance and Evolution · Stack Overflow · Refactoring

1 Introduction

Refactoring is a disciplined technique and a fundamental activity in software development. From the most structurally simplistic task of renaming an identifier to more complex structural changes such as extracting and moving a method, developers refactor their code to improve the internal quality of their systems while still preserving the system's behavior [56]. According to its definition, refactoring is applied to enforce best design practices or to cope with design defects. Thus, academic research has been recommending refactoring to either fix code and test smells (e.g., long methods, God classes, etc.) [55,63] or improve structural metrics (e.g., coupling, cohesion, cyclomatic complexity, etc.) [51,48]. However, recent surveys have shown the lack of adoption of refactoring tools in practice [71,62], in part due to the lack of support for the types of problems developers face when refactoring. One potential problem is that while refactorings tend to be applied alongside other development and maintenance tasks [76,30], automated refactoring tools do not fully take this aspect into account. In addition, this interleaving of actions indicates that there may be some relationship between refactoring and other development tasks, meaning that these other tasks are

Anthony Peruma is the corresponding author.

Anthony Peruma, Steven Simmons, Eman Abdullah AlOmar, Christian D. Newman, Mohamed Wiem Mkaouer
Rochester Institute of Technology
Rochester, New York, USA
E-mail: axp6201@rit.edu, sds9278@rit.edu, eman.alomar@mail.rit.edu, cnewman@se.rit.edu, mwmvse@rit.edu

Ali Ouni
ETS Montreal, University of Quebec
Montreal, Quebec, Canada
E-mail: ali.ouni@etsmtl.ca

essential for comprehending the refactoring. Unfortunately, there is still a lack of thorough understanding of the rationale behind applying a refactoring or the problems developers face when determining which refactorings to apply and how to apply them in practice.

Prior studies that have examined the rationale behind the application of refactoring activities relied on the examination of project artifacts such as commit logs (focusing mainly on a selected set of open-source projects) [80, 77, 27, 81] or interviewing a selected set of developers [62, 92]. However, interviews are a limited resource and may not fully generalize due to the necessarily limited number of people they gather information from. In addition, the natural language analysis of commit logs is not always a feasible approach because developers typically use high-level descriptions; rarely mentioning specific refactorings or project-specific reasoning for why changes have been made beyond high-level reasoning, such as improving readability/comprehension of the code, eliminating code/design smells, and fixing defects [80, 81, 71]. As a result, prior work has highlighted the challenges in correlating rationale obtained from various software artifacts with refactoring activities [80, 81, 71]. For instance, while we know that developers refactor to improve readability, it is still unclear how a developer identifies a readability issue, how the developer plans to perform this improvement, and how we can accurately obtain this information from software artifacts that are updated in response to these changes.

There are other sources of information, besides commit messages and interviews, for understanding the rationale behind refactoring; the most popular programming-specific question and answer forum on the internet, Stack Overflow, is one of them. With over 19 million questions [9] and one million users (as of January 2019) [18], Stack Overflow provides developers with a mechanism to seek advice and help from the community on a wide range of software development topics. Hence, through this study, we are presented with a diverse and representative view of developer discussions around refactoring; thereby, gaining a real-world understanding of not only the volume of discussions around refactoring but also the topics that are of most interest and challenging to developers.

1.1 Motivation

The field of software refactoring research is continually evolving. For instance, the initial definition of refactoring indicates that the driver to refactor source code arises from the degradation of the internal quality of a system [67]. However, through empirical [28] and developer-based [92] studies, it is apparent that code smells may not be the only reason developers apply refactorings. Given that it is common for academic research to recommend refactoring based on the presence of code smells, there is a clear need for a stronger understanding of the other drivers of refactorings so that appropriate recommendations can be made in those contexts as well. Quote 1 is an example of a developer seeking help with a code change task that was caused as a consequence of several factors: high complexity metrics and poor readability. They found some advice pointing at the Strategy design pattern as a solution, but are unsure of whether it is overkill for this problem or not. In this context, several studies have proposed solutions to fix brain methods or reduce cyclomatic complexity. However, little is known on how to address a problem that contains both of these characteristics. Therefore, there is a need to investigate techniques that can approach these types of issues holistically. Prior work has tried to fill this gap by interviewing and surveying developers. However, these studies are still limited in how much they can tell us about the general population of developers. The question is: *Are developers applying refactorings in the same environments, on problems with the same characteristics and context, as researchers assume?* To answer this question, we need to understand refactoring as it is discussed in day-to-day development activities. Thus, we focus our attention on real-world challenges developers currently face when refactoring their systems. For instance, consider the example in Quote 2. The majority of refactoring-related research focuses on statically typed programming languages, such as Java. However, dynamically typed languages such as Python and other software artifacts such as databases and configuration files may also need to be part of some refactorings, such as the rename refactoring.

Refactoring switch statement for Data to different types of data

"My mission is to refactor a switch statement that was poorly written (it makes the cyclomatic complexity spike). In short, there is a class that parses a file for various values. I have read up on the topic and while there seems to be plenty of help, it all seems to be pointing at the Strategy design pattern, which (I believe) would overkill my problem. In my project, there are multiple classes like this, with some of them having upwards of 25 case statements."

Quote 1: A developer runs into challenges trying to improve the quality metrics of the code. [14].

What refactoring tools do you use for Python?

"I have a bunch of classes I want to rename. Some of them have names that are small and that name is reused in other class names, where I don't want that name changed. Most of this lives in Python code, but we also have some XML code that references class names... Does anyone have experience with Python refactoring tools ? Bonus points if they can fix class names in the XML documents too."

Quote 2: An example of a refactoring challenge a developer faces when refactoring Python code. [21].

As one of the premier online question and answer resources for developers, Stack Overflow is an ideal candidate to mine for real-world discussions around programming-related challenges. The diversity of the user base presents us with an opportunity to study what and how developers refactor their systems effectively. The content we mine from Stack Overflow will assist us in gauging the extent to which current research in refactoring is lacking and understand the reasons for gaps between the application of automated refactoring and the way developers perceive it [92]. Furthermore, as this study captures the state of real-world refactoring at this current point in time, it becomes an ideal candidate for future replication-based studies to determine the extent to which real-world refactoring has evolved, such as the technologies and challenges developers face when refactoring their systems.

1.2 Goal & Research Questions

The goal of this study is to *understand the trends and challenges around developer discussions on software refactoring concepts and activities*. We envision our findings used as input by practitioners, researchers, and educators in understanding the current state of refactoring trends and challenges and determine the extent to which traditional (or academic) viewpoints of refactoring need revising based on real-world applicability. Additionally, IDE and tool vendors that offer refactoring support will find our results helpful in improving their instruments to better support developers. Hence, we first start by examining the volume of refactoring posts and user contributions. Through this examination, we reveal how popular and valuable Stack Overflow is in the developer community in discussing refactoring topics. Once we establish this volume, our next task is to determine the topics of discussion. We achieve this in three steps by analyzing the natural language text in the body of a question. We first identify common phrases; then, we look for specific refactoring terminology in the body. Finally, our third step is grouping similar questions based on the terms they utilize and then determining the category (or topic) of these groupings. The final analysis of this study utilizes the identified categories to determine the type of questions that are challenging to answer. Thus, we define and address the following research questions:

RQ$_1$: How have refactoring discussions on Stack Overflow grown over the years? Through this research question, we gain insights into the growth in refactoring related discussions throughout the years. This question explores the volume of questions and answers on Stack Overflow, along with the community members responsible for creating these posts. Additionally, we examine the tags that accompany refactoring questions, which presents us with high-level information into the areas that involve refactoring.

RQ$_2$: What do developers discuss in refactoring based Stack Overflow posts? This research question utilizes natural language techniques to identify the key terms and phrases developers utilize in crafting refactoring questions. Our analysis presents a real-world and granular view of the problematic or challenging software refactoring areas developers require assistance.

RQ$_3$: Which topics are the most popular and difficult among refactoring-related questions? This research question presents us with the opportunity to understand the refactoring topics that are popular among developers and topics that are difficult to answer on Stack Overflow. Additionally, we also examine unanswered questions.

1.3 Contributions

This study provides the community of researchers and practitioners with insights on the discussion of refactoring on Stack Overflow. More specifically, are contributions are outlined below:

- Mining and extraction of 9,489 Stack Overflow questions related to refactoring.

- A series of quantitative and qualitative experiments on the extracted questions to show the growth and trends of refactoring discussions between developers, such as:
  - The frequently occurring set of tags and terminology in questions, which also include an analysis of the use of refactoring specific terminology.
  - The primary topics for refactoring questions, including the popular and challenging topics, along with an analysis of unanswered questions.

We also make available our dataset for replication and extension purposes [13].

2 Related Work

Before discussing the research works related to our study, we first present a brief overview of the state of research in the field. Since developers discuss multiple aspects of refactoring on StackOverflow, we believe it is essential that readers comprehend the field's evolution to understand the extent to which academia address practical challenges developer face. The spectrum of research exploring the practice of refactoring covers a wide variety of dimensions. One of the earliest studies, by Mens and Tourwe [67], provides an overview of existing research in the field of software refactoring. They discuss the existing literature in terms of refactoring activities and techniques, refactoring tool support, and the impact of refactoring on the software process. Further studies on refactoring focus on studying the impact of refactoring on quality (e.g., [70,42,29,75,47,106]), identifying refactoring opportunities (e.g., [54,74]), recommending refactoring operations (e.g., [68,43,73]), and implementing refactoring tools (e.g., [84,66,61,103,91,69]).

As our study is around developer discussions on refactoring, our discussion of related work is limited to studies investigating the rationale and motivations of why developers refactor code or research into discussions around tools/technologies that developers rely on for refactoring code. To this extent, we group our set of related works into two groups. The first group contains studies that mine Stack Overflow posts and report on refactoring-based discussions among developers. The second group of studies is based on developer interviews or analysis of project artifacts to understand developer refactorings.

2.1 Refactoring Related Discussions in Stack Overflow Posts

Pinto and Kamei [82] mine Stack Overflow posts to study discussions around refactoring tools. From a tool features perspective, the authors observe that developers prefer tools that provide refactoring recommendations and support for database and multi-language refactoring. Additionally, findings from this study show that in addition to usability issues, the lack of trust in tools is one of the major barriers to adoption. In their study on code smells and antipattern discussions on Stack Overflow, Tahir et al. [99] observe that the majority of answers to these questions did not provide refactoring recommendations; instead, the answers provide details around the code smell/antipattern. Furthermore, developers do not frequently refer to refactoring operations by name in the posts and refer to some design patterns as potential refactoring solutions. In a subsequent study [98], the authors performed a large-scale study that explores how developers discuss code smells and antipatterns in Stack Exchange. The authors show that most of the questions focus on the following code smells: Duplicated Code, Spaghetti Code, God Class, and Data Class. As for the programming languages, most of the discussions focus on popular languages like C#, JavaScript, and Java. Although Java has greater tooling support, other platforms such as C# and JavaScript are lacking in support. Findings by Tian et al. [102], from their study on architecture smells, show the lack of tools for refactoring architecture smells. The authors also highlight that even though there exist specialized tools to refactor architecture smells, these tools are not mentioned in the Stack Overflow posts; instead, developers mention the use of common code smell detection tools. Additionally, the authors also observe that time and costs involved with detecting and refactoring architecture are very concerning for most developers. In a preliminary study specializing in the refactoring of Java 8 streams for parallelization, Tang et al. mined Stack Overflow posts for discussions around Java 8 streams [100]. As part of their findings, the authors mention that 5% of questions around Java 8 streams remain unanswered. A preliminary study by Choi et al. [49] on code clones shows that most discussions are related to refactoring with the need for more support for clone refactoring tools. Openja et al. [72] utilize topic modeling to study release engineering questions. More specifically, the authors examine popular topics among developers and those that are difficult to answer. In this study, the authors identify 38 topics from which questions around security are both challenging and popular.

While our study also mines Stack Overflow posts, the key difference between our study and the works described above is that our study is not limited to a specific programming language, paradigm, or technology. We retrieve and analyze any post related to the concept of refactoring.

2.2 Refactoring

Arnaoudova et al. [36] surveyed 71 developers to understand the importance of rename refactoring operations. The findings show that developers consider renaming refactoring a challenging activity, and they frequently perform rename operations on the source code. In their study of utilizing the commit log to contextualize renaming operations, Peruma et al. [80] observe developers perform renames as part of addressing defects and unit tests. The authors also highlight that the commit log alone cannot be utilized to gain insights into renaming operations. In a preliminary study on the refactorings of Android apps, Peruma [77] performs a topic modeling analysis on the commit log. The author shows that developers refactor apps for reasons such as improving code readability, fixing defects, and enhancing the app's design. A survey with developers at Microsoft by Kim et al. [62] shows that there are costs and risks involved with the performance of refactoring activities and also the need for more tool-based support. Furthermore, the survey results show that developers do not consider refactoring to be confined to only behavior preserving transformations; this is in contrast to the academic definition. In their study, Danilo et al. [92] identify that changes in requirements are one of the key reasons that drive developers to refactor code. The authors also show that 'Extract Method' is a frequently occurring refactoring operation and identify 11 motivations for applying this operation. Additionally, the authors also identify that developers are concerned about introducing duplicate code. Finally, the authors also indicate that developers more frequently apply refactorings manually than using a tool. They also report that a lack of trust in tools is a key concern among developers. Murphy-Hill et al. [71] note that the majority of refactoring operations are performed manually. However, the rename operation is frequently performed using a tool. Additionally, the authors also indicate that there exist instances where developers utilize tools to perform refactorings in batches. Examining commit messages, the authors show that it is not feasible to determine if a refactoring was applied based on the message in the commit log.

More recently, Pantiuchina et al. [76] present a mining-based study to investigate why developers perform refactorings by analyzing the history of 150 open-source systems. Particularly, they analyze 551 pull requests containing refactoring operations and produce a refactoring taxonomy that generalizes existing literature. In a large-scale empirical study on refactoring, AlOmar et al. [30] explore what motivates developers to apply refactorings by mining and automatically classifying a set of 111,884 commits containing refactoring activities extracted from 800 open-source Java projects. Their findings show that fixing code smells is not the main driver for developers to refactor their code. Developers refactoring for various reasons (e.g., feature addition, bugfix), going beyond its traditional definition. Furthermore, recent studies [29, 75] show that there is a misperception between the state-of-the-art structural metrics widely used as indicators for refactoring and what developers consider to be an improvement in their source code. The authors identified (among software quality models) metrics that align with the vision of developers on the quality attributes they explicitly state they want to improve. A number of studies have recently focused on the documentation of refactoring. AlOmar et al. [27, 28, 30] have explored how developers document their refactoring activities in commit messages; this activity is called Self-Affirmed Refactoring (SAR). In particular, SAR indicates developers' explicit documentation of refactoring operations intentionally introduced during a code change. Based on their empirical investigation, developers tend to use a variety of textual patterns to document their refactoring activities, besides '*refactor*', such as '*redesign*', '*reorganize*', and '*polish*'. These patterns can be either (1) generic, providing a high-level description of the refactoring, or (2) specific by explicitly mentioning the rationale behind the applied refactoring operations.

3 Study Methodology & Dataset Construction

Our research methodology follows a mixed-methods approach, where we collect and analyze both quantitative and qualitative data [101]. This approach provides insight into relationships between qualitative and quantitative data and allows us to present representative samples from the dataset to complement our findings. More specifically, our approach utilizes well-established statistical measures, including unsupervised machine learning and natural language processing techniques which we apply to our dataset to report on trends and patterns of refactoring posts. Additionally, we also manually review a statistically significant sample of refactoring posts (body and metadata) to gain further insight into developers' refactoring challenges to supplement and correlate our quantitative findings.

Figure 1 outlines the methodology for our study. Our methodology involves three key activities– (1) obtaining a recent and representative Stack Overflow dataset, (2) identifying and extracting refactoring posts, (3) analyzing the refactoring posts (textual content and metadata). In summary, our study utilizes *SOTorrent* [38] to obtain Stack Overflow posts. From this dataset, we extract all refactoring posts (based on the tag and title of the post) and analyze these discussion posts via both manual inspection and automated mechanisms to help answer our research questions. In the following subsections, we describe in detail the elements and activities that were part of our methodology. Furthermore, we have the complete dataset available on our website for replication and extension purposes [13].

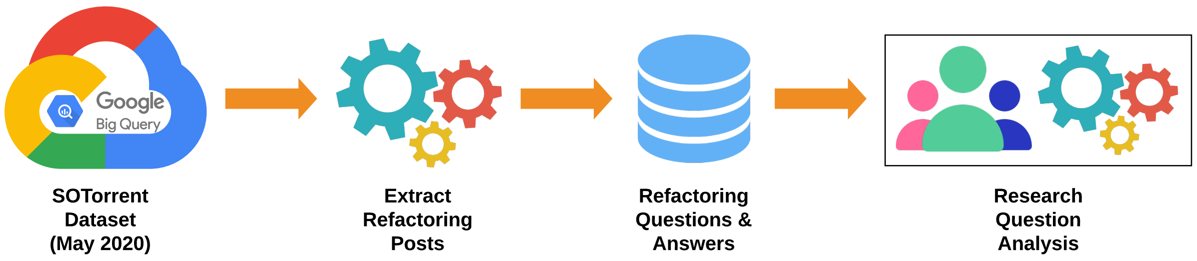

Google
Big Query
SOTorrent Dataset (May 2020)
Extract Refactoring Posts
Refactoring Questions & Answers
Research Question Analysis

Fig. 1: Overview of our study methodology.

3.1 SOTorrent

For our study, we utilized the March 2020 release of the *SOTorrent* dataset available on Google BigQuery[1]. The *SOTorrent* dataset is constructed using a data dump provided by Stack Overflow. The dataset contains the version history of each post along with other relevant metadata such as the author of the post, score, view count, answer count, etc. Provided below, we briefly describe the attributes in the dataset that we utilized in our experiments.

- **Posts**: There are three types of posts – (1) question, (2) answer, and (3) accepted answer. A question post contains details about the problem/challenge faced by the developer. A question consists of a title and body field; the title is a plain text field, while the body supports a limited set of formatting. Responses to a question can fall into one of two categories: an accepted answer or a non-accepted answer[2]. A response post contains only a body field. While a question can have only one accepted answer, it can have more than one non-accepted answer. An accepted answer indicates that the developer asking the question considers this specific answer as a solution to the question asked. In other words, an accepted answer acts as a means of informing the community that the specific answer was the intended solution for the question. Only the developer asking the question can mark an answer as an acceptable answer.
- **Tags**: As part of creating a question, the developer needs to associate the question with one to five tags– a word or phrase that describes the question[3]. Only questions can be associated with tags. Tags permit site users to access a particular set of questions that is of interest to them. Stack Overflow discourages the creation of arbitrary tags and instead recommends the use of predefined tags.
- **Score**: Associated with posts, this metric is based on the Upvotes the post receives. The higher the score value, the more useful the post to the community[4].
- **View Count**: Associated with only questions, this metric corresponds to the number of times the post was viewed[5].

[1] `https://bigquery.cloud.google.com/dataset/sotorrent-org:2020_03_15`
[2] `https://stackoverflow.com/help/accepted-answer`
[3] `https://stackoverflow.com/help/tagging`
[4] `https://stackoverflow.com/help/privileges/vote-up`
[5] `https://meta.stackexchange.com/questions/90187`

– **Favorite Count**: Associated with only questions, this metric indicates the number of times site users have marked the post as a favorite.

3.2 Refactoring Posts

Even though *SOTorrent* contained all Stack Overflow posts, our research focused on refactoring related discussions. To this end, we performed an extraction of relevant question-based posts from *SOTorrent*. Our process involved retrieving questions that had either been tagged with a word containing the term *'refactor'* or contained the term in the title. For each extracted question, we also extracted all answer-based posts (including accepted answers) associated with the question. We excluded searching for the term *'refactor'* in the body of the post as we observe that such an action leads to an increase of false positives in our dataset. Looking at such posts, we observe developers mentioning the term *'refactor'* in passing, even though the post is not about the actual refactoring of code. For instance, in question– Quote 3, the developer requests help to solve a runtime exception without refactoring the code. Additionally, in some instances, the developer refactors the code to make it easier to comprehend the problem [16]. Furthermore, as mentioned by Rosen and Shihab [86], the title succinctly describes the primary purpose of the question.

"The problem I have is that for a series of incoming messages...which leads to exceptions within the DLL...Is there any way around this, given that refactoring the DLL isn't an option."

Quote 3: A false positive refactoring example [8].

3.3 Results Analysis

Our study of refactoring posts includes a quantitative and qualitative approach. Our quantitative approach involves executing database queries and custom code/scripts, including a topic modeling analysis using an unsupervised machine learning algorithm. In the qualitative approach, two or more of the authors manually analyzed a statistically significant sample set of the data. Depicted in Figure 2, we summarize the type of research approach we utilize to answer each research question and the data on which it is applied. In Section 4, we elaborate in detail on our analysis approach to answering each research question.

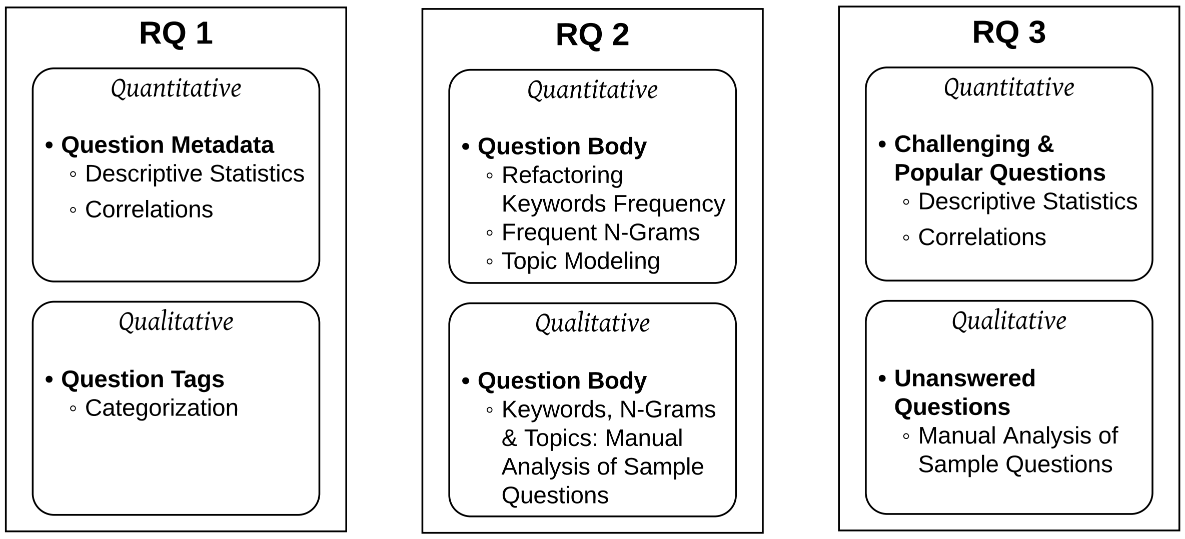


Fig. 2: Summary of the research approach we utilize to answer each research question (RQ).

Table 1: Summary of collected data.

Item	Count
Number of questions	9,489
Number of accepted answers	6,112
Number of non-accepted answers	14,491
Number of questions without an answer	828
Number of questions with a 'refactor' tag	7,024
Average number of answers per question	2
Average number of tags per question	3

4 Empirical Study Design & Results

In this section, we report on the results of our experiments. Our analysis contains three Research Questions (RQs), with certain RQs comprising of sub-RQs. In RQ_1, we look at the growth of refactoring posts, the individuals that ask and respond to questions, respectively, and the tags associated with questions. RQ_2 looks at identifying the rationale behind the refactoring posts. Specifically, the RQ examines the frequently occurring phrases in the body of a post and groups questions into categories (or topics) based on the textual content of the question. Finally, in RQ_3, we utilize the results from the prior RQ to identify popular and challenging topics.

For each RQ, we first explain the primary motivation(s) and the approach we undertake to produce the results; then, we present our findings. Further, where applicable, we also include examples to posts on Stack Overflow to provide the reader with more clarity on our observations. Finally, even though some tables and figures in the RQs show a subset of the data, we have the complete dataset available on our website for replication and extension purposes [13].

4.1 RQ_1: How have refactoring discussions on Stack Overflow grown over the years?

This RQ is composed of three sub-RQs exploring the growth of refactoring posts on Stack Overflow over the years. As a question and answer site, $RQ_{1.1}$ examines the growth of refactoring-based questions and answers on the site. In $RQ_{1.2}$, we examine if a selected set of Stack Overflow users is responsible for the majority of refactoring-related questions and answers. Finally, in $RQ_{1.3}$, we look at the tags used by developers in creating refactoring questions and the growth of these tags throughout the years.

4.1.1 $RQ_{1.1}$: How have refactoring posts grown throughout the years?

Motivation & Approach:

In this sub-RQ, we examine the growth of refactoring questions on Stack Overflow. The purpose of this sub-RQ is to understand the extent to which developers require help and advice on refactoring related problems and how often they receive the assistance they seek. To do this, we extract all questions that had the term 'refactor' in either the title or tag. Next, for each question, we extracted all answers (i.e., accepted and non-accepted) associated with the question.

Findings:

In total, we extracted 9,489 refactoring-related questions, from which, 828 (∼8.73%) of the questions did not have an associated answer. Regarding accepted answers, 6,112 (∼64.41%) of the questions had an accepted answer. Table 1 provides a summary of the collected data.

Figure 3 shows a yearly breakdown of refactoring questions with and without an accepted answer. In this chart, for each year, the orange bars are questions without an accepted answer, while the blue bars are questions with an accepted answer. The gray bars represent all questions (i.e., questions with and without an accepted answer). Our count of questions with an accepted answer is constrained to question-accepted answer pairs created in the same year. It should be noted that Stack Overflow was launched in September 2008, and our dataset contains posts up to March 2020; hence our analysis excludes posts created in these two years. A first glance at this chart shows that, other than for the year 2019, the number of questions with an accepted answer outnumber questions without an accepted answer. However, also shown in this

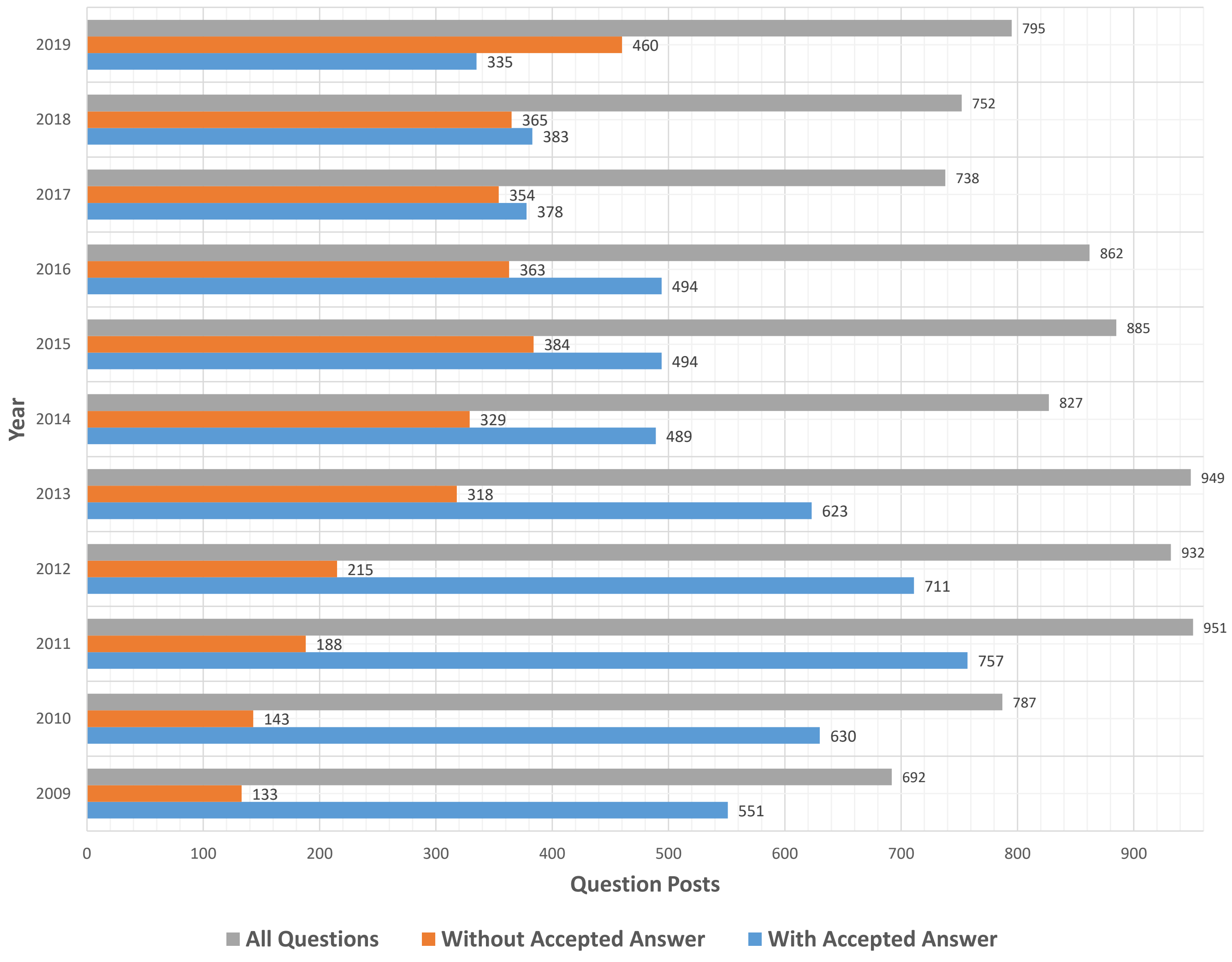


Fig. 3: Count of refactoring questions and accepted answers, per year.

chart is that as the years progress, the number of questions with accepted answers decreases while questions without accepted answers increases.

Next, we look at the time duration between a developer asking a question and receiving a response. In this analysis, we look at how long it takes for a question to receive a response (regardless of it being an accepted answer) and how long it takes to receive an accepted answer. When examining the time duration between question-accepted answer pairs, we consider all refactoring posts in our dataset and do not restrict our analysis to pairs created in the same year. For completeness, we present the median values with and without outliers (removed via the Tukey Fences approach [58]). The median time between a question and its first answer is 0.27 hours with outliers and 0.19 hours without outliers. The median time between a question and an accepted answer is 0.41 hours with outliers and 0.27 hours without outliers. Additionally, in Figure 4, we also show histograms of time duration values (without outliers)[6]. This chart shows that most responses to refactoring questions occur within the first hour of the developer asking the question.

Looking at the year-over-year (YOY) growth around refactoring discussions (refer to Figure 5), we see that questions and accepted answers share a similar pattern between the YOY growth for questions and accepted answers. While the number of questions and accepted answers have increased, the volume by which they increased has been falling. Furthermore, to measure the extent of the relationship between these two variables, we conducted a Pearson correlation test [97]. This particular test is also known as a parametric correlation test as it depends on a normal distribution of the data, which we confirmed via a

[6] When reading/comparing these two histograms, it should be noted that the 'Frequency' scale for the two charts differs.

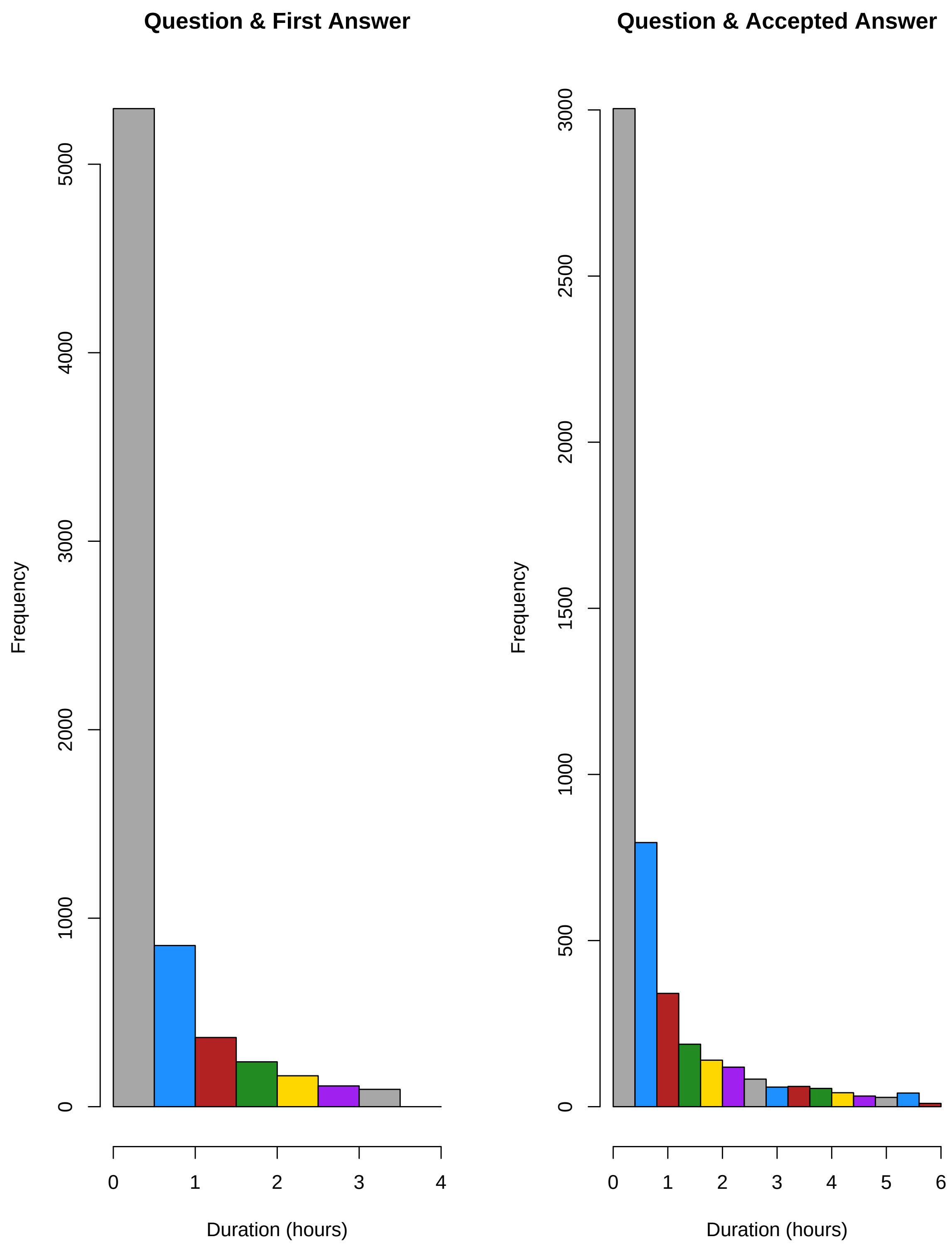
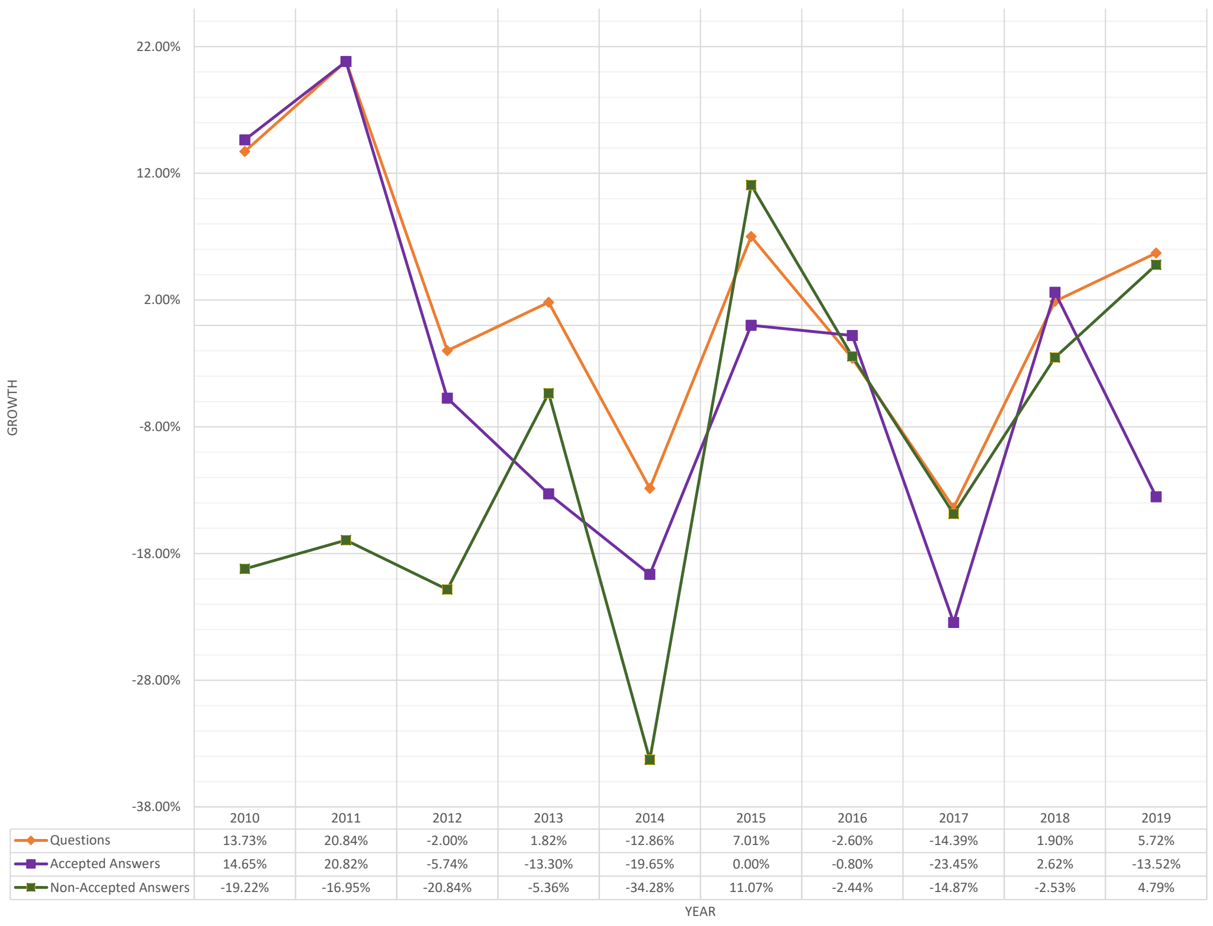


Fig. 4: Distribution of time (in hours) from when a question is asked until the reception of an answer.



	2010	2011	2012	2013	2014	2015	2016	2017	2018	2019
Questions	13.73%	20.84%	-2.00%	1.82%	-12.86%	7.01%	-2.60%	-14.39%	1.90%	5.72%
Accepted Answers	14.65%	20.82%	-5.74%	-13.30%	-19.65%	0.00%	-0.80%	-23.45%	2.62%	-13.52%
Non-Accepted Answers	-19.22%	-16.95%	-20.84%	-5.36%	-34.28%	11.07%	-2.44%	-14.87%	-2.53%	4.79%

Fig. 5: Year-over-year growth of refactoring posts.

Shapiro-Wilk normality test [97]. The Pearson correlation test yielded a statistically significant (i.e., p-value < 0.05) correlation coefficient of 0.872, equating to a strong correlation.

Finally, from the two charts, we observe a phenomenon where the number of questions forms a wave-like pattern throughout the years. Hence, while we see a dip in questions in 2017, there has been a gradual uptick in the subsequent years. This is an interesting phenomenon and would require further research to explain this pattern.

4.1.2 $RQ_{1.2}$: What is the distribution of questions and answers among developers?

Motivation & Approach:

After looking at the general trend of refactoring questions, this sub-RQ looks at the involvement of the Stack Overflow community in refactoring discussions. We investigate if a selected set of community members is responsible for asking and answering refactoring questions. To answer this sub-RQ, we capture the unique members posting questions and answers. More specifically, we utilize the `OwnerUserId` field to identify the creator of a post (i.e., question and answer).

Findings:

In the first part of our analysis, we look at the volume of unique users posting questions and answers. Looking at questions, we observe that 7,795 distinct users are responsible for creating all 9,489 refactoring-related questions. When it comes to answers, we found that 4,610 distinct users are associated with accepted answers, while 10,566 distinct users are associated with non-accepted answers.

Next, as shown in Table 2, we look at the distribution of these unique users for each type of post. From the whole set of distinct users posting questions, most ($\approx$ 82.64%) of the users post only questions. A similar pattern emerges when we look at the distinct set of users posting non-accepted answers; most

Table 2: Distribution of distinct user types.

Item	Count	Percentage
Questions - total distinct user: 7,795		
Users who post only questions	6,442	82.64%
Users who post questions & accepted answers	553	7.09%
Users who post questions & non-accepted answers	800	10.26%
Accepted Answers - total distinct user: 4,610		
Users who post only accepted answers	2,707	58.72%
Users who post accepted answers & questions	553	12.00%
Users who post accepted & non-accepted answers	1,350	29.28%
Non-Accepted Answers - total distinct user: 10,566		
Users who post only non-accepted answers	8,416	79.65%
Users who post non-accepted answers & questions	800	7.57%
Users who post non-accepted & accepted answers	1,350	12.78%

Table 3: Frequency distribution of the number of posts created by a user.

Number of posts created by a user	Count	Percentage
Questions - total distinct user: 7,795		
1	6,913	88.69%
2	599	7.68%
3	149	1.91%
4	65	0.83%
6	20	0.26%
Others	49	0.63%
Accepted Answers - total distinct user: 4,610)		
1	3,948	85.64%
2	425	9.22%
3	98	2.13%
4	53	1.15%
5	27	0.29%
Others	59	1.28%
Non-Accepted Answers - total distinct user: 10,566)		
1	8,856	83.82%
2	1,059	10.02%
3	316	2.99%
4	129	1.22%
5	58	0.55%
Others	148	1.40%

(≈ 79.65%) of these users only post non-accepted answers. However, looking at the set of distinct users posting accepted answers, we see more of an even distribution.

In Stack Overflow, it is possible for the user who asks a question to answer their question and even mark their answer as an accepted answer. Therefore, we next look at such instances for refactoring posts. When it comes to accepted answers, we observe that 6.50% (or 389) of question-accepted answer pairs have the same user creating the question and accepted answer. For non-accepted answer-question pairs, approximately 2.30% (or 322) instances have the question creator also posting an answer.

Our final analysis looks at the volume of posts that a user creates. We observe that the majority of distinct users asking questions would ask, at most, only one question (≈ 88.69%). We observe similar patterns for accepted and non-accepted answers. In these two instances, approximately 85.64% and 83.82% of the distinct users create a single accepted and non-accepted answer post, respectively. Table 3 shows the top five distributions for each type of post.

4.1.3 $RQ_{1.3}$: What are the tags that are associated with refactoring questions?

Motivation & Approach:

Tags are an essential part of Stack Overflow posts; they enable developers to categorize their questions and also help experts quickly access the questions that they specialize in. In this sub-RQ, we explore the types of tags that developers associate with refactoring questions. Our analysis of tags will help determine the concepts and technologies associated with refactoring challenges developers face. These findings provide a high-level overview, which we elaborate on in the subsequent RQs. To obtain the data for this sub-RQ, we extract all tags from all refactoring posts[7].

Findings:

In total, our dataset contains 3,053 distinct tags. Not surprisingly, the most frequently occurring tag in our dataset is ‘refactoring’. We observe 6,808 ($\sim$21.89%) questions containing the ‘refactoring’ tag. Since this study is on refactoring, going forward, our analysis of tags will exclude counting the ‘refactoring’ tag and instead focus on the other tags added by developers to refactoring posts to understand the area of refactoring better.

The top five tags were all related to programming languages (or web frameworks)– Java (1,529 or 7.58% instances), C# (1,512 or 7.50% instances), JavaScript (946 or 4.69% instances), Ruby on Rails (591 or 2.93% instances), and Ruby (569 or 2.82% instances). These top five tags accounted for 27.82% of the tags in the dataset. Additionally, most of the frequently occurring programming language tags in our dataset also appear in the list of top programming languages from 2016 to 2018 [93,94,95,96].

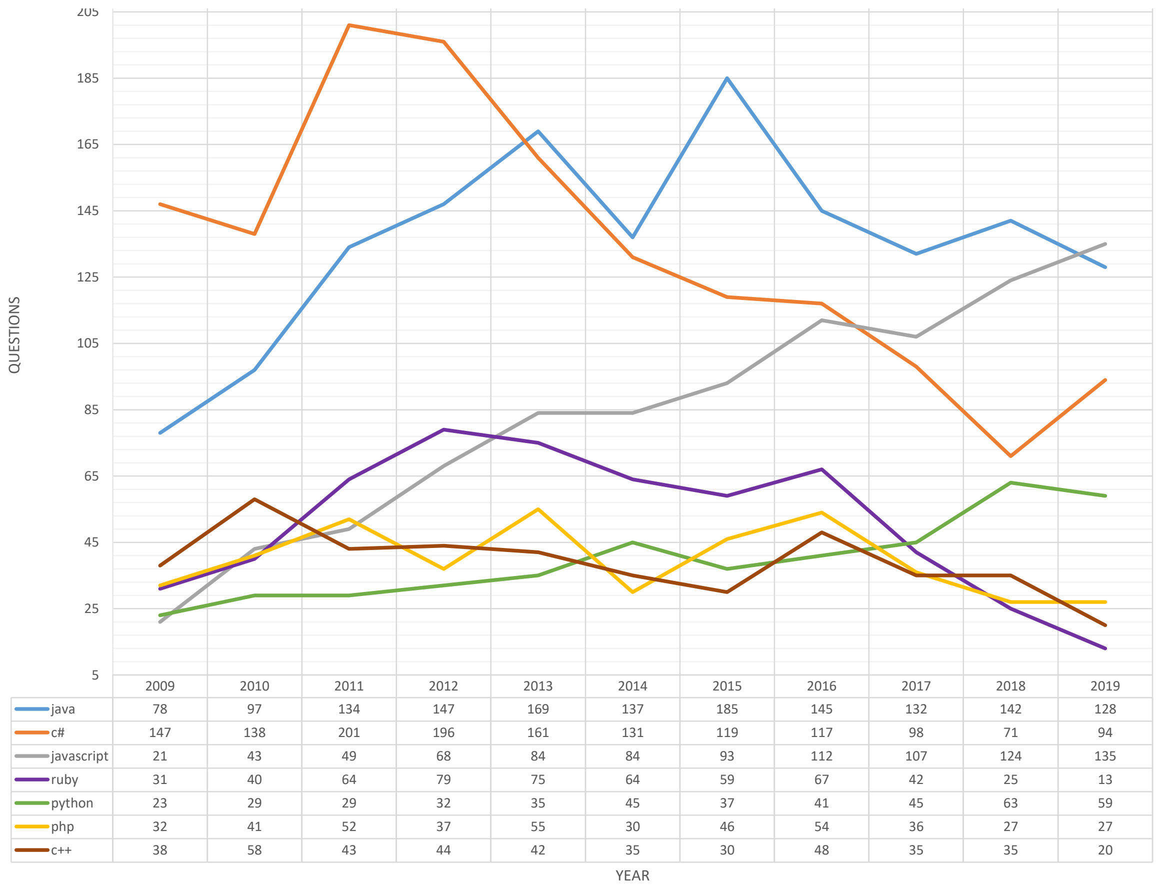
Next, as part of our analysis, we look at the yearly growth of the six most popular programming languages refactoring tags in the dataset. When reviewing the tags for each year, we observe that these tags were part of a frequently occurring set of tags each year. In Figure 6, we plot the volume by which questions associated with each tag either grow or shrink year-by-year (we ignore the years 2008 and 2020 as data is not available for the entire year). From this graph, we observe questions tagged with C# show a steep decline from 2012, while at the same time, we see the volume of Java tagged posts being more-or-less constant. We also observe a constant increase in JavaScript tagged posts throughout the years; similarly, though relatively low in volume compared to C# and Java posts, we also observe a steady increase in Python tagged posts. Furthermore, our findings on the rise of dynamically typed languages and the fall of statically typed languages are also in alignment with the Popularity of Programming Language Index [12].

To determine the different categories these tags fall under, we manually reviewed a statistically significant sample of tags. To this extent, two authors reviewed 547 of the frequently occurring tags; this sample corresponds to a confidence level of 99% and a confidence interval of 5%. When reviewing the utilized tags, each author made notes for each tag. Comparing notes, the authors discussed and settled on the finalized set of annotation categories. The finalized set consists of seven categories– Tools, Programming Languages, Framework/Library/API, Algorithms and Programming Concepts, Design/Architecture, Operating Systems, and Other. Presented in Figure 7, is a breakdown of the volume of instances for each tag category.

The majority of the tags (236 instances or 43.22%) are related to general algorithms and programming concepts, which include code quality (e.g., ‘code-cleanup’), data types (e.g., ‘arrays’), and programming concepts (e.g., ‘recursion’) among others. The next most popular category is frameworks/libraries/APIs (118 instances or 21.61%). Some of the tags under this category include javascript-based frameworks and libraries (e.g., ‘reactjs’), web frameworks (e.g., ‘django’), non-web libraries such as testing (e.g. , ‘junit’), object-relational mapping (e.g., ‘hibernate’), and others (e.g., ‘pandas’). Tags falling under the tools category (88 instances or 16.12%) include questions around IDEs (e.g., ‘eclipse’), database (e.g., ‘mysql’), and plugins (e.g., ‘resharper’), among others. The majority of the tags under programming languages (56 instances or 10.26%) include those that support a multi-paradigm model such as ‘java’, ‘c#’, and ‘python’; the next most common paradigm is declarative (e.g., ‘sql’ and ‘html’). The design/architecture category includes tags related to patterns (e.g., ‘model-view-controller’). The popular tags in the operating system category are mobile-based (i.e., ‘android’ and ‘ios’). Finally, the tag ‘iphone’ is the most popular tag in the other category, containing six instances or 1.10%.

Summary for RQ$_1$. Stack Overflow is a popular venue for developers to receive solutions for their refactoring questions, usually receiving a response in a short timeframe. Furthermore, while most questions involve statically typed languages, specifically Java and C#, we see a rise in questions around dynamically typed languages such as JavaScript and Python. This finding shows a need for refactoring support from researchers and tool developers for dynamically typed languages and their unique issues. Finally, our tags analysis shows that most questions are around algorithm and programming concepts, followed by questions around specific frameworks/libraries.

[7] In Stack Overflow, tags can only be associated with a question post.



	2009	2010	2011	2012	2013	2014	2015	2016	2017	2018	2019
java	78	97	134	147	169	137	185	145	132	142	128
c#	147	138	201	196	161	131	119	117	98	71	94
javascript	21	43	49	68	84	84	93	112	107	124	135
ruby	31	40	64	79	75	64	59	67	42	25	13
python	23	29	29	32	35	45	37	41	45	63	59
php	32	41	52	37	55	30	46	54	36	27	27
c++	38	58	43	44	42	35	30	48	35	35	20

Fig. 6: Yearly growth of the popular tags associated with refactoring posts.

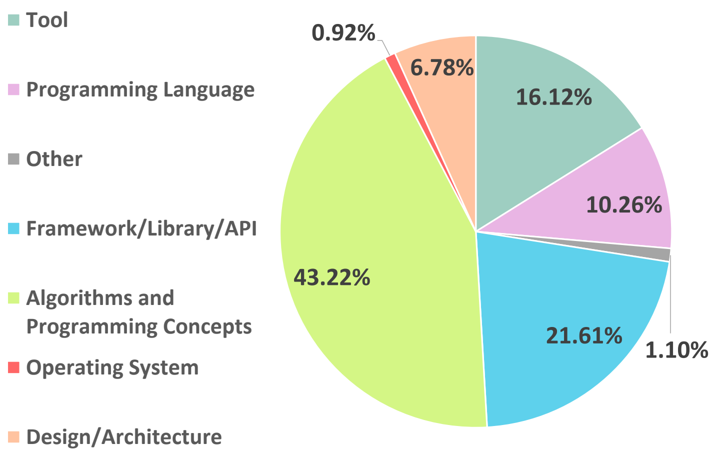


Fig. 7: Breakdown of the volume of instances for each tag category.

4.2 **RQ$_2$: What do developers discuss in refactoring based Stack Overflow posts?**

As a question and answer site, the fundamental purpose of Stack Overflow is to facilitate developers to utilize natural language to either describe the problem they require help with or provide advice/solutions to these problems. Prior work has shown that the language used to describe refactorings can vary heavily [27,28]. For instance, the word 'refactoring' is not always used (e.g., clean-up is a common alternative). Furthermore, the context surrounding the refactoring being applied can affect the particular alternative phrase that a developer chooses to use instead of the term 'refactoring'. Thus, in RQ2, we explore the language used by developers to discuss refactorings through questions and answers. The outcomes from this question are meant to support a more robust analysis of refactoring rationale by revealing how developers discuss refactorings and what terminology frequently correlates in refactoring-related discussion topics. This RQ is composed of three sub-RQs. In RQ$_{2.1}$, we examine the set of frequently occurring keywords in refactoring posts. In the second part of the research question, we utilize a refactoring taxonomy, known as Self-Affirmed Refactoring (SAR) [27], to define refactoring opportunities. This taxonomy contains internal structural metrics (coupling, complexity, etc.), external quality attributes (code comprehension, readability, reusability, etc.), and code smells (duplicate code, long methods, etc.). Existing studies have shown the existence of posts discussing the removal of code smells [99,98], but little is known about how developers discuss the other types. Therefore, In RQ$_{2.2}$, we investigate the extent to which this taxonomy is triggering discussions around their refactoring. Finally, in RQ$_{2.3}$, we perform a deep dive into the questions asked by developers by grouping related questions.

4.2.1 RQ$_{2.1}$: What are the frequent terms utilized by developers in refactoring discussions?

Motivation & Approach:

The prior research questions show that developers do indeed encounter real-world challenges when refactoring their systems, and, at a high-level, our analysis of tags also indicates areas associated with refactoring challenges. However, since developers need to select the tags from a predefined list, we are limited in understanding more specific refactoring areas. Hence, in this sub-RQ, we extract the top keywords as bigrams from question posts. Bigrams correspond to a sequence of two adjacent words in a sentence. We did look at trigrams, but we could not locate sets of common terms. Unlike unigrams, bigrams provide a certain level of context for terms, which helps our analysis by reducing the chance of making false presumptions. Additionally, we also examine the existence of specific refactoring terms. These terms correspond to refactoring operations defined in Fowler's catalog of refactoring operations (e.g., extract method, move method, rename attribute, etc.) [56].

Findings:

In Table 4, we present the top ten frequently occurring bigrams in question posts. This table shows that the IDE 'visual studio' plays an important part in refactoring discussions. Since this particular IDE can support multiple types of programming languages, it has become popular among developers in implementing systems [20]. The mention of the IDE falls into two categories– (1) developers asking for assistance in performing a refactoring operation using the IDE, or (2) developers mentioning the IDE when providing context around the refactoring challenge they encounter. Furthermore, the bigram 'refactoring tool' also emphasizes the importance and reliance of tools and IDEs in refactoring activities. We also observe discussions around the refactoring of test suites showing that interest in refactoring source code among developers is not just limited to production code. The bigram 'legacy code' highlights a common reason why developers request support with refactoring. In the subsequent sub-RQ's (RQ$_{2.2}$ and RQ$_{2.3}$), through qualitative analysis, we further contextualize most of these bigrams and provide exemplar posts from where they were extracted.

Since the names of refactoring operations do not occur in the top ten frequently occurring bigrams list, we investigate if developers use these terms in their questions. These are the terms defined in Fowler's catalog of refactoring operations [56]. We observe that the bigram 'extract method' frequently occurs in question posts. However, we should also note that we also encounter instances where developers talk about extracting other types of identifiers, such as classes, interfaces, and variables. For move operations, once more, we observe that developers frequently mention the 'move method' operation. However, we also observe instances of moving classes and files. We also observe the terms 'common' and 'code' associated with the refactoring operation terms 'extract' and 'move', showing that developers discuss refactoring operations without using the standard terms defined in the catalog or refactoring operations. Looking at

Table 4: Top ten frequently occurring bigrams for question posts.

Bigram	**Count**	**Percentage**
visual studio	348	1.57%
unit test	253	1.14%
base class	164	0.74%
create new	160	0.72%
refactoring tool	155	0.70%
switch statement	146	0.66%
business logic	133	0.60%
design pattern	132	0.59%
legacy code	115	0.52%
add new	110	0.50%
Others	20,484	92.27%

inline operations, once more, methods were mentioned frequently for inlining. However, looking at rename operations, we observe more occurrences of files, classes, packages, and variables than methods. Finally, push-up and pull-down operations did not yield anything significant (similar to findings by Danilo et al. [92]).

This sub-RQ highlights common bigrams in refactoring posts to understand why developers utilize Stack Overflow for refactoring discussions. These bigrams are phrases associated with refactoring operations, showing that developers are aware of refactoring concepts. Going one step forward, in the next sub-RQ, we explore the use of other known software engineering refactoring phrases.

4.2.2 RQ$_{2.2}$: To what extent do traditional refactoring opportunities, known in existing literature, match with the challenges faced by developers in Stack Overflow posts?

Motivation & Approach:

The concept of Self-Affirmed Refactoring (SAR) introduced by AlOmar et al. [27,28] explores how developers document their refactoring activities such as the intent behind the refactoring type of operations performed. In their work, the authors identify recurring patterns in SAR commit messages and define three SAR categories, (1) internal quality attributes, (2) external quality attributes, and (3) code smells. In our work, we use these SAR terminology patterns in addition to other keywords related to code smells reported in literature [56,99] that belong to each of the three SAR categories as indicators of refactoring activity-related discussions. In other words, we string match SAR patterns in question posts to see the extent to which they contribute to challenges developers face when they refactor. We also extract the bigrams from question posts to better understand the context of SAR patterns and help us in our analysis. We are particularly interested in extracting the intent behind the refactoring in questions to capture what typically triggers developers to refactor their code.

Findings:

Table 5 depicts the list of SAR patterns, ranked based on their frequency, we identify in questions. We observe that developers frequently mention key internal quality attributes (such as inheritance, cohesion, etc.) and a wide range of external quality attributes (such as readability and performance), and a variety of code smells that might impact code quality. Upon closer inspection of the generic refactoring patterns, we notice that developers use a variety of patterns to discuss refactorings such as 'clean up' or 'redesign', although 'refactor' is the most used keyword (29.09%). Additionally, these patterns are mainly linked to code elements at different levels of granularity (e.g., 'add an attribute', 'create an interface', 'refactor the method'). Further, we observe that developers mention the motivation driving refactorings that are not restricted only to fixing code smells, as in the original definition of refactoring in Fowler's book [56]. We also observe that developers tend to report the executed refactoring operations using keywords such as 'extract' or 'rename'.

To improve the internal design, the optimization of dependency seems to be the dominant focus that is consistently mentioned (41.86%). It is apparent from some of the posts that developers intend to introduce best practices (e.g., the use of object-oriented design principles, the application of inheritance, polymorphism, and optimization of software quality metrics to reduce code complexity). Further, developers refactor the code to improve the dominant modularization driving forces (i.e., cohesion and coupling) to maximize intra-class connectivity and minimize inter-class connectivity.

Concerning external quality attribute-related questions, we observe the mention of refactorings to enhance nonfunctional attributes. Terms such as 'readability', 'efficiency', and 'performance' represent the developers' main focus, with 16.37%, 13.85%, and 11.52%, respectively. Although multiple studies [75,53, 32] have been analyzing code comprehension and using metrics to measure readability, there is no mention of these readability tools/models (i.e., [50,89,46,83]) in the questions. For instance, developers refactor the code to improve its reusability. More recently, AlOmar et al. [31] show that the number of methods significantly increases when developers refactor the code to improve reusability. Also, developers make changes such as extracting methods to improve testability as they test parts of the code separately. Developers also extract methods to improve code readability.

Finally, for code smell-focused refactoring questions, we observe that duplicate code represents the most popular anti-pattern that developers intend to refactor (28.75%). While there are also various tools for detecting and potentially refactoring code clones [60,87,66], we could not locate any reference to them. Also, developers perform refactorings to eliminate specific code smells (e.g., spaghetti code, long method, feature envy, etc.) that are known to deteriorate the quality of the source code. For example, based on our manual analysis, we observe that developers discuss performing 'Extract Method' refactoring to remove a code smell, which corresponds to a long method (i.e., a bad smell). Developers also indicate the generic pattern 'code smell' in addition to the specific name of the code smell under correction.

This sub-RQ shows that SAR patterns documented in commit messages are also utilized by developers when crafting questions on Stack Overflow. While these phrases/terms are indicative of refactoring activities, they are specific to SAR patterns and, at a high-level, do not provide context as to where or what is associated with the terms. Hence, in the following sub-RQ, we group all related terms (i.e., SAR and non-SAR pattern terms) to determine the primary categories that cause developers to seek assistance when refactoring. We perform this grouping on the entire set of questions on our dataset via the use of an unsupervised machine learning technique to determine the refactoring topics associated with refactoring questions.

4.2.3 RQ$_{2.3}$: What are the topics around software refactoring that are being asked by developers?

Motivation & Approach:

The prior sub-RQs looked at common and SAR pattern terms in refactoring posts. In this sub-RQ, we go a step forward by grouping related terms to identify and understand the different areas (or topics) that developers require assistance with and understand the motivation behind refactoring.

We tackle this research question from three fronts. First, we perform an n-gram analysis to identify the common phrases developers utilize when describing their problem/challenge. Next, we perform a topic modeling analysis to identify the key topics associated with refactoring related questions. Finally, we manually analyze a statistically significant sample of questions to gain more insight and context around the detected topics. 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 [57]. N-grams are sets of co-occurring words (or letters), within a given window, that are available in a textual document and are useful in understanding a word in its context [59].

Prior to our topic modeling and n-gram analysis, we perform a set of pre-processing activities on the textual data. Some of our key pre-processing activities included: expansion of word contractions (e.g., 'I'm' → 'I am'), removal of URLs, code blocks, stopwords, alphanumeric words, and punctuations, retaining only nouns, verbs, adjectives, and adverbs and lemmatization of words. We opted to use lemmatization over stemming, as the lemma of a word is a valid English word [64]. In addition to the default set of stopwords supplied by NLTK [44], we added our own set of custom stop words. To derive the set of custom stop words, we generated and manually analyzed the set of frequently occurring words in our corpus. Examples of custom stop words include 'thanks', 'question', 'answer', etc. We utilize the Latent Dirichlet Allocation (LDA) [45] algorithm for our topic modeling analysis. Our use of LDA for the topic modeling analysis follows prior research based on Stack Overflow posts [72,105,39,41,86,104,23,108,40,33,26,37,22] that have shown the effectiveness of LDA in similar contexts. Essentially, 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. As the topics are not labeled, subject matter experts are then needed to determine each topic's name based on the analysis of the list of words. A mandatory input for the LDA algorithm is the number of topics to be generated. A low value will result in high-level or general topics, while a high value

Table 5: Frequency of Self-Affirmed Refactoring (SAR) patterns in questions posts.

Generic SAR		**Specific SAR**		
Keyword	**Percentage**	**Internal QA**	**External QA**	**Code Smell**
refactor*	29.09%	dependency (41.86%)	readability (16.37%)	duplicate code (28.75%)
chang*	10.02%	complexity (14.06%)	efficiency (13.85%)	switch statement (19.23%)
creat*	9.12%	inheritance (13.27%)	performance (11.52%)	code smell (12.27%)
add*	8.30%	polymorphism (8.05%)	configurability (6.78%)	case statement (6.95%)
mov*	7.05%	coupling (7.74%)	productivity (6.08%)	redundancy (4.76%)
clean*	3.64%	abstraction (7.26%)	effectiveness (5.67%)	spaghetti code (4.02%)
renam*	3.61%	composition (4.73%)	maintainability (5.55%)	anti-pattern (4.02%)
remov*	3.27%	encapsulation (2.05%)	usability(4.73%)	long method (3.84%)
replac*	3.07%	cohesion (0.94%)	testability (4.44%)	dead code (3.66%)
extract*	2.71%		compatibility (3.85%)	large class (2.38%)
fix*	2.40%		reusability (3.33%)	god class (1.64%)
improv*	2.12%		modularity (2.22%)	technical debt (1.46%)
modif*	1.83%		flexibility (2.16%)	bad smell (1.46%)
simplif*	1.59%		accessibility (1.98%)	data class (1.46%)
rewrit*	1.41%		manageability (1.92%)	long parameter list (0.91%)
reduc*	1.38%		reliability (1.57%)	complex class (0.54%)
split*	1.26%		simplicity (1.40%)	middle man (0.54%)
extend*	1.05%		extensibility (1.16%)	code clone (0.54%)
introduc*	1.03%		accuracy (1.05%)	inappropriate intimacy (0.36%)
merg*	0.74%		stability (0.99%)	feature envy (0.36%)
pull*	0.71%		understadability (0.93%)	anti pattern (0.36%)
get rid of*	0.66%		robustness (0.70%)	brain class (0.18%)
migrat*	0.66%		scalability (0.64%)	message chain (0.18%)
organiz*	0.64%		correctness (0.40%)	
inlin*	0.49%		modifiability (0.23%)	
push*	0.41%		adaptability (0.11%)	
tidy*	0.19%		reproducibility (0.11%)	
restructur*	0.19%		repeatability (0.05%)	
customiz*	0.18%		interoperability (0.05%)	
redesign*	0.18%			
aggrega*	0.15%			
reformat*	0.13%			
reorganiz*	0.13%			
enhanc*	0.11%			
decompos*	0.09%			
rework*	0.08%			
modulariz*	0.04%			
refin*	0.03%			
polish*	0.02%			
re packag*	0.005%			

will produce more detailed topics, some of which will be noise. Hence, to arrive at the optimal number of topics, we iteratively extracted topics 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. In each cycle, we extracted the topic coherence [85], perplexity score [45], and topic visualization [90] of the model. Finally, to determine the optimal number of topics for our LDA analysis, we relied on a combination of topic coherence, perplexity, visualization, and manual analysis. The complete set of coherence and perplexity values for each of the fifty models and an interactive visualization for the optimum model is available on our project website. Concerning our manual and visual analysis– we look 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 manually examined the list of generated terms to determine the appropriate topic names. For our manual analysis, we undertook a collaborative approach– we looked at the terms that represent each topic, came to an agreement on the name of the topics, and identified the topics generated by noisy terms. Additionally, we also looked at the terms that are unique to each topic and the terms shared among topics (including the overall frequency of the term).

Table 6: LDA topics with their frequency of occurrence and a partial set of their corresponding words.

Topic	**Question**		**Key Words**
	Count	**Percentage**	**(unigrams & bigrams)**
Code Optimization	4,142	43.72%	loop, array, function, variable, operator, parameter, lambda, dictionary, repeated, change, switch_statement, helper_method, lambda_expression, avoid_duplication, global_variable
Tools and IDEs	2,395	25.28%	rename, package, resharper, file, tool, folder, import, ide, error, change, visual_studio, android_studio, separate_file, intellij_idea, command_line
Architecture and Design Patterns	1,660	17.52%	class, method, object, interface, model, abstract, subclass, inherit, singleton, factory, base_class, design_pattern, business_logic, derived_class, view_controller
Unit Testing	1,086	11.46%	test, unit, mock, logical, tdd, testable, coupled, coverage, junit, proxy, unit_test, business_logic, test_case, write_test, add_new
Database	190	2.01%	table, column, row, field, join, mysql, schema, dto, subquery, entity, stored_procedure, primary_key, sql_query, foreign_key, sql_server

Findings:

From our LDA analysis, we observe that the most optimal model yields five topics, associated with Stack Overflow refactoring questions: *Code Optimization*, *Architecture and Design Patterns*, *Unit Testing*, *Tools and IDEs*, and *Database*. To understand the distribution of each topic in the dataset, we assigned the most dominant topic to each question. Our results show that *Code Optimization* is the most frequently occurring topic (at 43.72% or 4,142 questions). We present, in Table 6, the distribution for each topic in the dataset. Additionally, the table also shows a partial set of words (unigrams and bigrams) associated with each topic.

While examining the words occurring in a topic helps in determining the name of the topic, it does not adequately help in determining the rationale for the topic. These topic-based words alone do not indicate the problems developers encounter or the advice they solicitor around refactoring. Hence, we perform a manual analysis of a stratified statistically significant set of questions. Using a confidence level of 95% and an interval of 10% for each topic, we constructed a sample size of 430 posts that were analyzed by two authors. In this process, two authors annotated the dataset with the rationale behind the topic. Next, the authors exchanged the annotated datasets for review. During the review, if the reviewer disagreed with a specific annotation, the instance was marked for discussion. Finally, the annotator and reviewer discussed and looked at resolving the identified conflicts.

For each of the detected topics, we describe our findings and summarize the key challenge(s) and the takeaways for relevant stakeholders. Additionally, to provide context around topics, we include representative examples to Stack Overflow posts in the form of quotes.

4.2.3.1 Code Optimization

Program comprehension is a crucial-enough concern for developers that they turn to the community for assistance with improving the analyzability or readability of their source code. To this end, a common challenge developers face is reducing lines of code. Our analysis of exemplar posts (e.g., Quote 4) shows that developers seek assistance with performing refactoring operations involving code extraction and advice on any patterns that they should follow. Specifically, some common challenges include simplifying or replacing switch statements and loops, compacting logic, and removing duplicate code. In most instances, developers are dealing with code that consists of a series of complicated logic conditions that require significant and careful refactoring so as to not result in a break of functionality. Within this topic, we also observe situations where developers reach out to the community for help with resolving runtime and behavioral issues they encounter after making readability improvements to their code. Thus, showing that improving program comprehension is not always straightforward– it can be both time-consuming and error-prone.

Next, observing the frequent terms in this topic (refer to Table 6), we encounter terms related to readability, such as ‘repeated’, ‘helper method,’ and ‘avoid duplication’; further highlighting that developers seek assistance with improving code reusability as a means of improving overall program comprehension. More specifically, this improvement often leads to optimization of switch-case statements and eliminating duplicate code, and possibly even improve the efficiency and performance of the system. Finally, it is worth noting that even though developers seek help with refactoring their code, they rarely utilize established refactoring terminology (e.g., 'Extract Method') when describing their problem; instead, they tend to be more colloquial in their description.

How can I refactor this Python code to make it more readable and compact?

"I wrote a function to handle selecting and using items to regain the player's health in a text adventure. What would be the best way to shrink the following code? Any feedback or constructive criticism would be greatly appreciated."

Quote 4: Sample question for the code optimization topic highlighting the need for assistance with improving program comprehension [5].

In summary, this topic shows that developers primarily seek assistance with simplifying code structures to improve readability, and reusability, specifically around reducing the complexity caused by lengthy switch-case statements, loops, and duplicate code. This challenge presents the research community with two important opportunities: 1) to conduct studies around the automatic detection and refactoring of lengthy conditional code blocks (such as switch-case statements), and 2) to understand and measure the influence of different code structures, patterns, and architectures on comprehension (i.e., what is the best way to refactor a large group of nested conditional statements?). This challenge is related to prior work, which shows that readability metrics are currently struggling to measure real readability [53].

4.2.3.2 Tools and IDEs

This topic deals with questions about activities related to tool-based refactoring. Tools are a vital part of software development. They provide developers with the means to automate time-consume code changes, thereby improving developer productivity and potentially eliminating the injection of defects. Predominately, questions around this topic are related to renaming activities. This aligns with the previous topic where we observe developers seeking assistance to optimize their code to improve program comprehension. However, questions around tool-based renaming are not just related to a straightforward renaming of a source code identifier. Instead, developers seek assistance with renaming other software engineering artifacts such as packages, database elements, files, and content within other non-source code files such as XML or performing bulk/batch-based renaming operations (e.g., Quote 5). In most instances, the questions are around using the rename functionality of their IDE.

While most questions we observe are around renaming, we also encounter questions about performing other refactoring operations such as move operations, type changes, duplicate checking/elimination. Once again, these questions are specific to the developer's project and, in most cases, nontrivial. Additionally, developers seek advice for recommendations for tools/IDEs for refactoring specific programming languages or for configuring tools (such as disabling/enabling specific IDE refactoring features). In their study of the usability of refactoring tools, Eilertsen and Murphy [52] highlight the need for tools to support developers in guiding tools in the execution of refactoring operations. This corroborates our findings, where most questions around tool usage are specific to a developer's source code. Furthermore, many of the standard refactoring operations available by the tool do not meet the developer's unique refactoring needs. Lastly, from Table 6, we observe that questions involve using popular IDEs, specifically Visual Studio, IntelliJ IDEA, and Android Studio.

Rename/Refactor database elements - only scripts exists but not database

"I have script files ready to create a database. I also have coding standards/conventions to be set against those scripts. Is there a best/Easy way to rename these (according to coding standards I have) such that when I rename a database object, other objects that reference the renamed object should automatically updated with the new name."

Quote 5: Sample question for the tools/IDEs topic showing developers seeking assistance using tools/IDEs to perform nontrivial rename operations [15].

In summary, this topic shows that developers understand the benefits of using tools to automate refactoring activities. However, the developer's expectation goes beyond the general/standard features offered by the tools. The primary challenge is with seeking assistance with renaming other software engineering artifacts outside of identifier names. These are artifacts such as packages, database elements, files, and others. Based on this, vendors should enhance their IDE's renaming facility to support renaming other software elements/artifacts (e.g., database elements) related to renamed source code identifiers. Furthermore, the research community needs to investigate models and techniques that can learn from and adapt to a developer's codebase. Such as providing better recommendations and improved detection of opportunities to refactor artifacts outside of code. Especially those artifacts that have direct links to entities in the code that are touched by a refactoring.

4.2.3.3 Architecture and Design Patterns

As part of their evolution, industrial systems typically undergo architecture refactoring to maintain their structural quality [88]. A review of a sample set of posts on this topic shows that reusability is a crucial motivator for developers to refactor their systems. Developers primarily seek assistance with reducing the complexity of their systems, which in most accounts are legacy systems. Typically such systems have been online for a considerable period and undergone many updates by multiple developers. To this extent, the questions revolve around asking for assistance with adhering to design principles such as SOLID, DRY, SRP, and KISS. When asking for assistance, developers knowingly admit that their codebase violates specific design principles such as the existence of large modules (i.e., low cohesion) and duplicate or near-duplicate code. For instance, in Quote 6, the developer realizes that a method in their class performs more actions than it should and seeks advice on how best to refactor the method to adhere to the single responsibility principle. Such code negatively impacts both program comprehension and unit testing. Also, similar to the other topics, there exist situations where structurally based refactoring leads to issues to which developers turn to the community for assistance.

By following well-established architecture/design principles and patterns, developers construct their system using concrete and well-tested solutions and at the same time promote code reuse, performance, reliability, and extensibility. Additionally, the vocabulary of the pattern conveys the purpose of the pattern and thereby aids in communication between developers. Finally, Table 6 shows some of the common patterns developers mention in their post (e.g., 'singleton', 'factory', and 'view controller').

Can this MVC code be refactored using a design pattern?

"I've got controller code like this all over my ASP.NET MVC 3 site... we have caching, user reputation handling, auditing, all in one. Doesn't really belong in one spot does it. Hence the problem with the current code, and the problem with trying to figure out how to move it away."

Quote 6: Sample question for the architecture/design patterns topic where a developer requires assistance with making their code more robust by adhering to the single responsibility principle [3].

In summary, this topic shows that as a system evolves, the accumulation of updates made to the source code often results in the structure of the codebase violating established design principles and patterns. Developers face challenges refactoring their code to revert these violations and seek assistance from the community around applying SOLID, DRY, SRP, and KISS principles to their codebase. For the research community, this presents an opportunity to investigate how to provide stronger advice to developers attempting to stick to best practices. One potential avenue is to look at heuristics or AI model-based approaches to pointing out violations of, for example, DRY.

4.2.3.4 Unit Testing

Unit testing is an essential part of ensuring the quality of a system. Similar to production code, developers also refactor their test cases to meet internal quality requirements and/or to support the refactored production code. In most instances, developers face challenges with writing test cases, primarily to accommodate refactored production code. Since there can be more than one test method to evaluate a single production method, an update to the structure of production code usually involves extensive updates to the test suite. For example, in Quote 7, the developer runs into issues with the test suite after refactoring the system under test (i.e., production code) and seeks advice from the community on the appropriate approach that needs to be followed to update test cases.

Furthermore, it should be noted that while we do encounter questions around test configuration and API usage (e.g., mocking), the majority of the questions revolve around the developer's specific project code. Finally, the presence of the terms complexity, maintainability, and testability in the Self-Affirmed Refactoring patterns list (refer to Table 5) shows that developers recognize the importance of writing code that is test friendly. By reducing the cohesiveness and cyclomatic complexity of their production code, developers will find it much easier to construct test cases that achieve a high degree of modularization (i.e., responsibility) and code coverage.

TDD and Refactoring the "system under test"

"There were requirements changes and I had to change some classes behavior and API Changing one class behavior eventually led to changing a few others. I didn't know how to start this process from the test side, so I started changing the code. I ended up with lots of compilation errors in the test code and after I fixed them some did not pass. But the thing is, I don't even know if the tests cover what they used to cover before... TDD is supposed to give me a safety net while refactoring. Isn't it? As it currently appears in my case it doesn't give me that."

Quote 7: Sample question for the unit testing topic where a developer requires assistance updating the test suite due to production code refactoring [19].

In summary, test cases, like production code, are subject to evolution during the system's lifetime. However, evolving a test suite alongside production code can be challenging, and developers seek assistance incorporating these changes into the test suite. For instance, there can be multiple test methods (i.e., test cases) associated with a single production method. Hence, the refactoring of a single production method will require multiple updates to the test suite. The research and vendor communities can support developers with tools that either give recommendations or automatically refactor test suites when developers refactor the system under test.

4.2.3.5 Database

Traditionally refactoring has been primarily focused on improving the quality of program source code. However, these are not the only software engineering artifacts that developers refactor in real-world systems [67]. One such artifact is a database. A database is a crucial part of most software systems, and a highly optimized database ensures better performance in terms of speed and resource utilization, among other attributes [34].To this extent, most questions in this topic revolve around the refactoring of SQL queries to resolve challenges around performance and maintainability. In terms of performance, developers look for help reducing memory and query execution time by optimizing queries to return only the required set of records. From a maintainability perspective, improving analyzability or program comprehension is a crucial concern among developers. Like optimizing source code, developers seek assistance with reducing the complexity and length of queries. As an example, in Quote 8, the developer seeks assistance to improve the performance of a query and at the same time indirectly states that the complexity and readability of the query needs improving. Further, we also observe developers seeking assistance with performing batch/bulk rename operations to database elements. Finally, developers also require help improving the reusability or modularization of queries by removing duplicate code and adherence to the single responsibility principle.

Performance issue - refactor select subqueries which are doing the same joins

"I am getting performance issues because in the subqueries, I need to join on the same tables for each subqueries which is an heavy operation. Consider the below (ugly) example... so to me it sounds like the joins in subqueries is overkill even if it is working fine (very slow...)"

Quote 8: Sample question for the database topic where a developer requires assistance improving the performance and maintainability of a SQL query [10].

In summary, from this topic, we observe that developers prefer to implement business logic within SQL scripts and these queries tend to grow in length and complexity. As such, this negatively impacts code readability, design principles, and system performance. The research/vendor community should provide developers with tools that automatically refactor or suggest changes to database elements based on source code refactoring and vice versa. Additionally, research into the readability of SQL scripts will lead to metrics and tools that developers can utilize in their implementation workflow.

Table 7: Popularity and difficulty of refactoring topics.

Topic	*Popularity Metrics*			*Difficulty Metrics*				
	Average Counts			**Questions w/o an ans.**		**Questions w/o an accepted ans.**		**Median hrs. to an accepted ans.**
	Views	**Favorites**	**Score**	**Count**	**Pct.**	**Count**	**Pct.**	
Code Optimization	168.2	1.09	0.81	277	6.69%	1,289	31.12%	0.23
Tools and IDEs	450.9	1.61	1.81	316	13.19%	1,014	42.34%	0.40
Architecture and Design Patterns	247	1.14	1.29	122	7.35%	594	35.78%	0.32
Unit Testing	289.23	1.67	1.33	96	8.84%	406	37.38%	0.34
Database	218.3	1.26	1.00	17	8.95%	69	36.32%	0.44
Refactoring Average	*274.73*	*1.35*	*1.25*	*165.60*	*9.00%*	*674.4*	*36.59%*	*0.34*

Summary for RQ$_2$. Our analysis of refactoring discussions shows questions revolving around five topics– *Code Optimization*, *Tools and IDEs*, *Architecture and Design Patterns*, *Unit Testing*, and *Database.* The primary driver behind these questions is the need to improve non-functional quality attributes in the code, of which improving maintainability is a key concern. Improving readability (such as reducing lengthy conditional statements) and reusability is of utmost concern for developers and is not only related to source code. Furthermore, synchronizing refactoring changes across software engineering artifacts (such as unit tests and databases) is also challenging for developers. Additionally, we highlight takeaways under each topic for the key stakeholders.

4.3 RQ$_3$: Which topics are the most popular and difficult among refactoring-related questions?

Motivation & Approach:

The purpose of this RQ is to understand the types of refactoring questions asked by developers that are challenging to answer. Additionally, we also look at the type of questions that the Stack Overflow community considers popular (or attractive). To this extent, this RQ builds on the results of the prior RQ to understand the specific refactoring topics that are considered popular or challenging by the community.

Our study of popularity and difficulty of topics is similar to prior research [108, 37, 22]. We measure the popularity of a topic by looking at the average view count, favorite count, and score of questions associated with each topic. The higher the average value for each metric, the more popular the topic. When it comes to topic difficulty, we look at the questions that do not have any answers, do not have an accepted answer, and the median time the community takes to provide an acceptable answer to a question. Since an answer can only be set as an accepted answer by the developer who asks the question, there can be situations where the person asking the question forgets to mark a provided answer as an acceptable answer. Hence, we look at the percentage of questions with no answers and those that have accepted answers.

Findings:

4.3.1 Popularity

From Table 7, we see that the topic *Tools and IDEs* is the most popular among the five refactoring topics while *Database* is the least popular topic. Even though the total number of *Tools and IDEs* questions are less than *Code Optimization* questions, the average view count metrics of the former are higher than the latter (by a percentage difference of approximately 91.33%). This indicates that developers are more frequently searching for, and thereby viewing, questions around refactoring tools and IDEs, perhaps, looking for help with a similar problem they are experiencing.

Additionally, we also look at the same metrics for all non-refactoring Stack Overflow questions (i.e., questions that are not part of our refactoring dataset) with the goal of comparing the popularity of refactoring questions against all other types of questions. In general, non-refactoring Stack Overflow questions have an average view count of 2361.42, an average favorite count of 0.62, and an average score of 2.08. Looking at the two sets of values, we see that refactoring questions have more favorites than general questions. At the same time, the view count is much higher for general Stack Overflow questions than refactoring

Table 8: Pearson correlation analysis between the popularity and difficulty of refactoring topics.Bold values indicate a statistically significant correlation (i.e., p-value < 0.05).

coefficient / p-value	**Views**	**Average Favorites**	**Score**
% w/o any answer	**0.921/0.026**	0.753/0.142	0.816/0.092
% w/o accepted answer	**0.950/0.013**	0.750/0.144	**0.949/0.014**
Hrs. to accepted answer	0.453/0.443	0.424/0.477	0.423/0.478

questions (a percentage difference of approximately 158.313%). The score metric for both these two types of questions is similar.

4.3.2 Difficulty

Once more, looking at Table 7, we observe that the topic *Tools and IDEs* has the most number of questions without both an accepted answer or any answer for that matter, and takes around 0.40 hours to receive an accepted answer. Hence, this seems to be the most challenging type of question for developers to answer. Topics falling under *Code Optimization* are less challenging to answer, as only 6.69% of these questions do not have an answer, while 31.12% of the questions do not have an accepted answer. Furthermore, it takes around 0.23 hours for such questions to receive an accepted answer.

4.3.3 Popularity & Difficulty Correlation

Similar to the prior work mentioned above, we perform a correlation analysis of the three popularity metrics (i.e., average view, favorite, and score) against the three difficulty metrics (median time to obtain an accepted answer, percentage of questions without any answers, and without an accepted answer). A Shapiro-Wilk normality test [97] on these variables shows that the data follows a normal distribution; therefore, we Pearson correlation test [97]. Table 8 shows the results of our correlation analysis. This table shows a strong positive statistically significant correlation (i.e., p-value < 0.05) for the difficulty metric percentage of questions without an accepted answer and the popularity metrics average views and score. Additionally, there is a strong positive relationship between views and posts without any answers. The remaining popularity and difficulty metrics do not show any statistically significant correlations. From this, we see that questions without an accepted answer are considered to be difficult, yet interesting enough that they garner views and scores from the developer community. This phenomenon is observable with the topic metrics in Table 7.

4.3.4 Unanswered Questions

In our final analysis, we examine unanswered questions (i.e., questions without an accepted and non-accepted answer post). To this extent, two of the authors manually reviewed a stratified statistically significant sample of 259 unanswered questions. This sample represents a confidence level of 95% and an interval of 10% for each topic, from a total of 784 unanswered questions. As part of the review, the authors examined the unanswered questions for ambiguity, incompleteness, or lack of concrete examples (source code, diagrams, etc.) to determine the lack of developer interaction with these questions.

The examination of these questions reveals that a majority were not ignored per se but received comments instead of answer posts. The content of these comments, in most situations, provides the author of the question with high-level suggestions on addressing the question or requests more clarification about the question. Since the volume of content permitted in a comment is restricted when compared to an answer post[8], most suggestions in comments were brief and tend to provide hyperlinks to other resources such as other Stack Overflow questions/answers, API documentation, and blog posts (e.g., [17]). With regards to clarifications, we observe developers utilizing comments as a means to have a back-and-forth discussion for clarity on the problem faced by the developer (e.g., [4]). Additionally, in some comments, the respondents simply state that there is no solution for the developer's issue (e.g., [6]) or is a known or reported bug (e.g., [1]). We also observe that a minority of unanswered questions are answered by the developer asking the question; the answer either appears as a comment or as an edit to the question (e.g., [7]). Finally, it

[8] `https://meta.stackexchange.com/questions/19756`

was interesting to note that there exist some unanswered questions that were not refactoring related (i.e., the developer misuses the refactoring term or tag in the question), showing that most of the developers understand the purpose of refactoring and its application (e.g., [2]). Hence, the majority of unanswered questions were not actually ignored by the community but responded to through comments.

Examining the questions' body, we observe that most of the questions around tools were asking for help in solving issues specific to the developer's project, such as altering the standard find-and-replace or refactoring operation for a particular purpose (e.g., [2]). We also observe that most questions around improving or implementing code reusability are often focused on identifying best practices [11], which is usually considered an opinion-based question by the Stack Overflow community. Such questions are often discouraged by the community as there is no commonly accepted answer and thus end up going unanswered even if they are comparatively coherent (i.e., it is clear what the author's problem and intent were). At times, such questions are suggested to be migrated to another site, most commonly Code Review Stack Exchange[9], an affiliated site, to be better addressed.

Summary for RQ$_3$. Questions around refactoring tools/IDEs are popular and challenging to answer, while questions around optimization of code snippets are the least difficult to answer. Questions that do not have an (accepted) answer usually have responses from the community as comments, which are usually high-level suggestions to address the issue or a request for clarification.

5 Discussion and Takeaways

Our findings show that Stack Overflow is a popular venue for developers to seek assistance with refactoring challenges for various technologies. Developers can post refactoring questions related to a range of technologies and artifacts, and usually receive a response in a short period of time. By performing a manual analysis of a statistically significant set of question posts, in our RQs, we supplement our quantitative findings and obtain an accurate understanding of the challenges developers face when refactoring their systems. Furthermore, our findings empower educators to update their course curriculum to reflect real-world settings better. For instance, 1) instilling the need for students to practice test-driven development in projects, 2) the importance of conducting early and frequent reviews of all types of software engineering artifacts, and 3) ensuring that these artifacts capture the non-functional goals of the system (especially around readability and reusability). In this section, through a series of takeaways, we discuss how our findings support the community.

Research Community Takeaways

While the research community has made considerable strides in refactoring related research, our findings demonstrate the challenges developers face in real-world projects. These findings highlight the gaps between the academic definition of refactoring and its actual usage in real-world settings. Furthermore, they provide the research community with opportunities to further evolve the field.

Adaptation of refactoring operations for multiple programming language and artifact types

Researchers have traditionally based their refactoring studies on statically typed programming languages (especially Java). However, our findings show that while this does benefit developers in real-world scenarios, there are opportunities for researchers to evolve the field further and increase the diversity of their research. To this extent, there is a need to adapt traditional refactoring operations to support dynamic language types (e.g., Python and JavaScript), which are rising in popularity. Further, the research community should also examine the possibility of deriving refactoring operations specific to dynamic languages. Additionally, programming source code files are not the only artifacts that developers associate with quality. Our findings show challenges with improving the quality around other related artifacts such as database elements (tables, columns, queries, etc.) and test suites, which are not frequently studied in research—further highlighting opportunities to evolve the refactoring field.

Improve and extend the applicability of readability quality metrics

[9] `https://codereview.stackexchange.com`

Our findings show that improvements to readability are a critical concern for developers. While the research community has made considerable strides in producing readability metrics and models [46,89], the community needs to better collaborate with established vendors in integrating their contributions with popular tools and IDEs to promote the usage of their artifacts. Additionally, our findings highlight specific avenues for readability research, such as optimizing/eliminating lengthy switch-case statements and conditional loops and understanding their influence on comprehension. Our findings also show that developers seek assistance with improving the readability of database artifacts, such as SQL queries and table/column names. While database vendors and the research community have provided developers with material to optimize the setup and performance of database systems, there is not much support around readability improvements to database elements. Developers specifically struggle with performing bulk renaming of database elements and improving the readability of long and complex queries.

Expand the study and applicability of reusability beyond source code

Along the lines of readability, developers also seek assistance with improving reusability, and like readability, the reusability assistance is not limited to source code. In addition to removing duplicate code from source code, developers also seek assistance with improving the reusability of database queries. To this extent, the research community should investigate refactoring at an architectural level to provide developers with information and recommendations around the structure of their codebase to improve reusability. Furthermore, there is also an opportunity to conduct research around reusability metrics and models for database artifacts.

Tool/IDE Vendor Community Takeaways

Like the research community, tool and IDE vendors play a vital part in ensuring developers write and maintain quality code. Hence, there are specific findings from our study which vendors can utilize to enhance their tools/IDEs. Our study shows that developers either mention the IDE/tool when providing context to their problem or ask questions specifically around the IDE/tool.

Automatic synchronization between project artifacts.

We observe a trend that while source code is central to refactoring, developers struggle with updating other artifacts (e.g., test cases, databases) due to refactoring of the source code. For instance, a single production method can be evaluated by more than one test method. Hence, refactoring of production code can result in multiple updates to the test suite. Likewise, renames to database tables/columns should cascade to SQL scripts and the data access layer in the source code. To this extent, IDE's should provide users with the ability to either synchronize refactorings across artifacts automatically or at the very least recommend refactoring opportunities in the related artifacts.

Enhanced rename refactoring functionality.

Our findings show that as most questions revolve around renames, IDE/tool vendors should consider incorporating the rename refactoring work by Peruma et al. [81,79], Liu et al. [65], Arnaoudova et al. [36], and Allamanis et al. [25] into their products to better provide developers with an automated approach to identifying, appraising and suggesting high-quality identifier names.

Enhance the user experience.

In addition to providing extensive and innovative refactoring functionality in their tools/IDEs, vendors must ensure that their products also exhibit an optimal user experience. Usability and trustworthiness are an essential part of refactoring tool adoption and are among the reasons for the lack of usage [71,52]. Our study corroborates these findings where we observe developers requiring help to configure tools or finding a tool for a specific purpose.

Developer Community Takeaways

Developers can also utilize our findings to ensure they follow a disciplined approach to software implementation. Our findings can be incorporated into the project's software development process as checklists/guidance that developers need to adhere to before certification of a release/deployment. Organizational managers can also utilize our findings to ensure that their development team is trained in the necessary skills required for refactoring and has access to the appropriate tools.

Extend coding standards utilized in projects to support naming standards for all project artifacts.

While it is common for project teams to utilize organizational or technology-specific coding standards, teams should ensure that the standards in use apply to all types of project artifacts utilized in the project. For instance, while general standards define the naming standards for source code identifier names (e.g., method names should begin with a verb), they do not define the naming standards for database table/column names. Furthermore, teams should also be made aware of the concept of linguistic anti-patterns [35] and how to detect and correct such occurrences in the code [78].

Integrating code quality tools into the build process for the early detection of poor coding practices.

Using code quality tools (e.g., code/design smell detectors [24]) during implementation will supplement the review tasks by automating the time-consuming task of detecting poor-programming practices (e.g., duplicate code, poor identifier naming, high cyclomatic complexity). Furthermore, there should be necessary checks in the project's process to ensure that developers can only ignore specific quality rules after providing valid justification.

Perform frequent and early peer-reviews on all project artifacts.

Peer reviews should not be limited to source code. Other artifacts such as architecture/design documents, test suites, and database artifacts should also undergo peer-reviews. Early reviews of such artifacts will ensure that the implemented system meets its non-functional goals. For instance, code reviews will help address readability issues before it accumulates to be a serious concern. Architectural and design artifacts will ensure that the system follows appropriate design principles, such as addressing reusability and modularization. Likewise, reviews of database queries will help in addressing readability and performance issues. Furthermore, these database-related reviews will also help to ensure the extent to which business logic is contained within SQL scripts (i.e., stored procedures) versus in the source code.

6 Threats To Validity

This section discusses the threats that may potentially impact the validity of our study. We group the treats into three categories– Internal, External, and Construct [107].

Internal Validity: *These are factors that influence our results.* We constructed our dataset by extracting and analyzing questions with the 'refactor' tag or contain the term 'refactor' in the title. There is the possibility that we may have excluded synonymous terms/phrases. However, even though this approach reduces the number of posts in our dataset, it also decreases false positives. Our approach ensures that we analyze posts that are explicitly geared towards refactoring challenges faced by developers. In other words, these are posts where developers were explicitly considering a refactoring action and were aware that they were attempting refactoring. Additionally, as the goal of this study is to understand the refactoring challenges faced by developers, our analysis is focused on the questions posted by developers. As with similar studies, our study is limited to analyzing only the most recent version of a post. Our analysis also does not take into account comments associated with a post, as comments are not considered as answers; comments in Stack Overflow are considered as temporary "Post-It" notes, and not every user has the privilege of creating a comment[10].

External Validity: *These are factors that impact the generalizability of our findings.* Even though there are several technology-based question and answer websites, our scope (and analysis) focuses exclusively on Stack Overflow– the largest such site on the Stack Exchange network[11] that caters to a wide range of computer programming topics. Furthermore, from RQ_1 (Section 4.1), we observe a large quantity of refactoring question and answer posts asked and answered by a diverse set of developers relating to various technologies. Additionally, the *SOTorrent* dataset, containing the Stack Overflow dump, has been widely used in similar knowledge sharing based studies. While we recognize that developer surveys/interviews are also viable mechanisms to study refactoring challenges, our study captures questions posted by 7,795 distinct Stack Overflow users. Furthermore, our findings provide either a starting or comparison point for future participatory-based research. Finally, while it is true that our findings capture the state of refactoring at the time we conducted our study, our results present snapshots of the data, which future studies can leverage to examine (e.g., via replication-based studies) the evolution of the field and also the state of Stack Overflow.

[10] `https://stackoverflow.com/help/privileges/comment`
[11] `https://data.stackexchange.com/`

Construct Validity: *Here we identify the extent to which our experiments are designed to measure what they are supposed to measure.* For RQ's that involves a qualitative analysis (such as annotations), we manually analyze a sample set of posts. However, these samples are statistically significant and followed a peer-review process to counter any bias in annotating. Further, since the review process involved a discussion between the annotators for each conflicting annotation, there was no need for calculating the inter-annotator agreement as the finalized dataset was conflict-free. The use of LDA in our topic modeling algorithm can be considered as a threat. However, as mentioned in the RQ (Section 4.2.3), this algorithm has been heavily utilized in similar studies. Furthermore, our selection of five topics is based on our analysis of fifty topics (in increments of one); our analysis included evaluating topic coherence, perplexity score, and visualization. Additionally, we also manually review a statistically significant sample of questions associated with each of the five topics. Finally, even though we utilize the Pearson correlation coefficient to measure the relationship between variables, there are other statistical measures, such as Cohen's d and ANOVA, which can also be applied to the data.

7 Conclusion and Future Work

Software refactoring is an essential activity in the maintenance and evolution of software. However, given the complexity of a system and the experience of the developer maintaining the system, performing refactoring operations can prove to be challenging. Hence, developers usually seek assistance from the community through question-and-answer websites such as Stack Overflow.

In this empirical study, we perform a quantitative and qualitative analysis of refactoring questions asked by developers on Stack Overflow. Our quantitative approach involved applying statistical measures on the mined data, while the qualitative analysis involved a manual review of a statistically significant sample of questions. Our results show that Stack Overflow is a popular online resource for developers to seek assistance with refactoring challenges. Our findings show that while most developers seek assistance with traditional statically typed languages (specifically Java and C#), there is a growing increase in refactoring dynamically typed code such as Python and JavaScript. Looking at the topics developers need assistance with, we observe that most questions are around optimizing source code to improve readability and reusability. However, source code is not the only artifact that developers refactor. Other artifacts, specifically database-related elements, are also subject to refactoring. Furthermore, developers find it challenging to propagate refactoring changes between related project artifacts. Tools are also a popular discussion topic among developers, specifically around the renaming of content within non-source code files and advanced refactoring automation. From our findings, we highlight a series of actionable takeaways for relevant stakeholders that will evolve the field of refactoring and improve developer productivity.

For our future work, we plan on conducting a structured survey with both junior and senior software developers from both open-source and industry. The survey will explore their general and specific challenges when performing refactoring activities; this includes (but is not limited to) software engineering artifacts, tools, and technologies associated with refactoring activities. This survey will complement and validate our current Stack Overflow study to provide the software engineering community with a more comprehensive view of refactoring practices.

8 Acknowledgments

We would like to thank the reviewers at ESE for their detailed and invaluable feedback.

References

1. Adding and removing additional documents with roslyn. `https://stackoverflow.com/questions/43933933`. (Accessed on 06/05/2020)
2. Automated refactor of myenum.myvalue.tostring() to nameof(myenum.myvalue). `https://stackoverflow.com/questions/51482063`. (Accessed on 06/05/2020)
3. Can this mvc code be refactored using a design pattern? `https://stackoverflow.com/questions/9597529`. (Accessed on 06/05/2020)
4. Complex refactor and version control with database projects. `https://stackoverflow.com/questions/29132487`. (Accessed on 06/05/2020)
5. How can i refactor this python code to make it more readable and compact? `https://stackoverflow.com/questions/56925926`. (Accessed on 06/05/2020)

6. How to refactor css grid for ie11 compatibility. https://stackoverflow.com/questions/53667530. (Accessed on 06/05/2020)
7. How to remove non-project files from refactorings in idea? https://stackoverflow.com/questions/49636340. (Accessed on 06/05/2020)
8. Making a thread-unsafe dll call in biztalk orchestration (or only running one orchestration at a time). https://stackoverflow.com/questions/7106884. (Accessed on 06/05/2020)
9. Newest questions - stack overflow. https://stackoverflow.com/questions. (Accessed on 06/14/2020)
10. Performance issue - refactor select subqueries which are doing the same joins. https://stackoverflow.com/questions/59950019. (Accessed on 06/05/2020)
11. Php refactoring, too many methods in class? https://stackoverflow.com/questions/31029037. (Accessed on 06/05/2020)
12. Popularity of programming language index. http://pypl.github.io/PYPL.html. (Accessed on 11/03/2020)
13. Project website. https://www.scanl.org/
14. Refactoring switch statement for data to different types of data - stack overflow. https://stackoverflow.com/questions/4299770/refactoring-switch-statement-for-data-to-different-types-of-data. (Accessed on 06/05/2020)
15. Rename/refactor database elements - only scripts exists but not database. https://stackoverflow.com/questions/46043341. (Accessed on 06/05/2020)
16. ruby on rails - how can i know which columns in my table are considered unique? - stack overflow. https://stackoverflow.com/questions/20929619. (Accessed on 06/05/2020)
17. Scala refactoring. https://stackoverflow.com/questions/33958310. (Accessed on 06/05/2020)
18. State of the stack 2019: A year in review - stack overflow blog. https://stackoverflow.blog/2019/01/18/state-of-the-stack-2019-a-year-in-review/. (Accessed on 06/14/2020)
19. Tdd and refactoring the "system under test". https://stackoverflow.com/questions/38334608. (Accessed on 06/05/2020)
20. Top ide index. https://pypl.github.io/IDE.html. (Accessed on 11/03/2020)
21. What refactoring tools do you use for python? https://stackoverflow.com/questions/28796. (Accessed on 06/05/2020)
22. Abdellatif, A., Costa, D., Badran, K., Abdalkareem, R., Shihab, E.: Challenges in chatbot development: A study of stack overflow posts. In: Proceedings of the 17th International Conference on Mining Software Repositories, MSR '20, p. 174–185. Association for Computing Machinery, New York, NY, USA (2020). DOI 10.1145/3379597.3387472
23. Ahmed, S., Bagherzadeh, M.: What do concurrency developers ask about? a large-scale study using stack overflow. In: Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM '18. Association for Computing Machinery, New York, NY, USA (2018). DOI 10.1145/3239235.3239524. URL https://doi.org/10.1145/3239235.3239524
24. Aljedaani, W., Peruma, A., Aljohani, A., Alotaibi, M., Mkaouer, M.W., Ouni, A., Newman, C.D., Ghallab, A., Ludi, S.: Test smell detection tools: A systematic mapping study. In: Evaluation and Assessment in Software Engineering, EASE 2021, p. 170–180. Association for Computing Machinery, New York, NY, USA (2021). DOI 10.1145/3463274.3463335. URL https://doi-org.ezproxy.rit.edu/10.1145/3463274.3463335
25. Allamanis, M., Barr, E.T., Bird, C., Sutton, C.: Learning natural coding conventions. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2014, p. 281–293. Association for Computing Machinery, New York, NY, USA (2014). DOI 10.1145/2635868.2635883. URL https://doi.org/10.1145/2635868.2635883
26. Allamanis, M., Sutton, C.: Why, when, and what: Analyzing stack overflow questions by topic, type, and code. In: Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13, p. 53–56. IEEE Press (2013)
27. AlOmar, E., Mkaouer, M.W., Ouni, A.: Can refactoring be self-affirmed? an exploratory study on how developers document their refactoring activities in commit messages. In: 2019 IEEE/ACM 3rd International Workshop on Refactoring (IWoR), pp. 51–58 (2019). DOI 10.1109/IWoR.2019.00017
28. AlOmar, E.A., Mkaouer, M.W., Ouni, A.: Toward the automatic classification of self-affirmed refactoring. Journal of Systems and Software p. 110821 (2020)
29. AlOmar, E.A., Mkaouer, M.W., Ouni, A., Kessentini, M.: On the impact of refactoring on the relationship between quality attributes and design metrics. In: 2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), pp. 1–11. IEEE (2019)
30. AlOmar, E.A., Peruma, A., Mkaouer, M.W., Newman, C.D., Ouni, A., Kessentini, M.: How we refactor and how we document it? on the use of supervised machine learning algorithms to classify refactoring documentation. Expert Systems with Applications p. 114176 (2020)
31. AlOmar, E.A., Rodriguez, P.T., Bowman, J., Wang, T., Adepoju, B., Lopez, K., Newman, C., Ouni, A., Mkaouer, M.W.: How do developers refactor code to improve code reusability? In: 19th International Conference on Software and Software Reuse, pp. 261–276. Springer (2020)
32. Alrubaye, H., Alshoaibi, D., Alomar, E., Mkaouer, M.W., Ouni, A.: How does library migration impact software quality and comprehension? an empirical study. In: International Conference on Software and Software Reuse, pp. 245–260. Springer (2020)
33. Alshangiti, M., Sapkota, H., Murukannaiah, P.K., Liu, X., Yu, Q.: 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), pp. 1–11 (2019)
34. Ambler, S., Sadalage, P.: Refactoring Databases: Evolutionary Database Design. Addison-Wesley Signature Series (Fowler). Pearson Education (2006)
35. Arnaoudova, V., Di Penta, M., Antoniol, G.: Linguistic antipatterns: what they are and how developers perceive them. Empirical Software Engineering **21**(1), 104–158 (2016). DOI 10.1007/s10664-014-9350-8. URL https://doi.org/10.1007/s10664-014-9350-8
36. Arnaoudova, V., Eshkevari, L.M., Penta, M.D., Oliveto, R., Antoniol, G., Guéhéneuc, Y.: Repent: Analyzing the nature of identifier renamings. IEEE Transactions on Software Engineering **40**(5), 502–532 (2014)

37. Bagherzadeh, M., Khatchadourian, R.: Going big: A large-scale study on what big data developers ask. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2019, p. 432–442. Association for Computing Machinery, New York, NY, USA (2019). DOI 10.1145/3338906.3338939. URL `https://doi.org/10.1145/3338906.3338939`
38. Baltes, S., Dumani, L., Treude, C., Diehl, S.: Sotorrent: reconstructing and analyzing the evolution of stack overflow posts. In: A. Zaidman, Y. Kamei, E. Hill (eds.) Proceedings of the 15th International Conference on Mining Software Repositories, MSR 2018, Gothenburg, Sweden, May 28-29, 2018, pp. 319–330. ACM (2018). DOI 10.1145/3196398.3196430. URL `https://doi.org/10.1145/3196398.3196430`
39. Bandeira, A., Medeiros, C.A., Paixao, M., Maia, P.H.: We need to talk about microservices: An analysis from the discussions on stackoverflow. In: Proceedings of the 16th International Conference on Mining Software Repositories, MSR '19, p. 255–259. IEEE Press (2019). DOI 10.1109/MSR.2019.00051
40. Bangash, A.A., Sahar, H., Chowdhury, S., Wong, A.W., Hindle, A., Ali, K.: What do developers know about machine learning: A study of ml discussions on stackoverflow. In: Proceedings of the 16th International Conference on Mining Software Repositories, MSR '19, p. 260–264. IEEE Press (2019). DOI 10.1109/MSR.2019.00052
41. Barua, A., Thomas, S.W., Hassan, A.E.: What are developers talking about? an analysis of topics and trends in stack overflow. Empirical Software Engineering **19**(3), 619–654 (2014). DOI 10.1007/s10664-012-9231-y. URL `https://doi.org/10.1007/s10664-012-9231-y`
42. Bavota, G., De Lucia, A., Di Penta, M., Oliveto, R., Palomba, F.: An experimental investigation on the innate relationship between quality and refactoring. Journal of Systems and Software **107**, 1–14 (2015)
43. Bavota, G., De Lucia, A., Marcus, A., Oliveto, R.: Recommending refactoring operations in large software systems. In: Recommendation Systems in Software Engineering, pp. 387–419. Springer (2014)
44. Bird, S.: Nltk: The natural language toolkit. ArXiv **cs.CL/0205028** (2002)
45. Blei, D.M., Ng, A.Y., Jordan, M.I.: Latent dirichlet allocation. J. Mach. Learn. Res. **3**(null), 993–1022 (2003)
46. Buse, R.P., Weimer, W.R.: Learning a metric for code readability. IEEE Transactions on Software Engineering **36**(4), 546–558 (2009)
47. Cedrim, D., Sousa, L., Garcia, A., Gheyi, R.: Does refactoring improve software structural quality? a longitudinal study of 25 projects. In: Proceedings of the 30th Brazilian Symposium on Software Engineering, pp. 73–82. ACM (2016)
48. Chávez, A., Ferreira, I., Fernandes, E., Cedrim, D., Garcia, A.: How does refactoring affect internal quality attributes? a multi-project study. In: Proceedings of the 31st Brazilian Symposium on Software Engineering, SBES'17, p. 74–83. Association for Computing Machinery, New York, NY, USA (2017). DOI 10.1145/3131151.3131171
49. Choi, E., Yoshida, N., Kula, R.G., Inoue, K.: What do practitioners ask about code clone? a preliminary investigation of stack overflow. In: 2015 IEEE 9th International Workshop on Software Clones (IWSC), pp. 49–50 (2015)
50. Dorn, J.: A general software readability model. MCS Thesis available from (http://www. cs. virginia. edu/weimer/students/dorn-mcs-paper. pdf) **5**, 11–14 (2012)
51. Du Bois, B., Demeyer, S., Verelst, J.: Refactoring - improving coupling and cohesion of existing code. In: 11th Working Conference on Reverse Engineering, pp. 144–151 (2004). DOI 10.1109/WCRE.2004.33
52. Eilertsen, A.M., Murphy, G.C.: The usability (or not) of refactoring tools. In: 2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 237–248 (2021). DOI 10.1109/SANER50967.2021.00030
53. Fakhoury, S., Roy, D., Hassan, S.A., Arnaoudova, V.: Improving source code readability: theory and practice. In: Proceedings of the 27th International Conference on Program Comprehension, pp. 2–12. IEEE Press (2019)
54. Fontana, F.A., Braione, P., Zanoni, M.: Automatic detection of bad smells in code: An experimental assessment. Journal of Object Technology **11**(2), 5–1 (2012)
55. Fontana, F.A., Mangiacavalli, M., Pochiero, D., Zanoni, M.: On experimenting refactoring tools to remove code smells. In: Scientific Workshop Proceedings of the XP2015, XP '15 workshops. Association for Computing Machinery, New York, NY, USA (2015). DOI 10.1145/2764979.2764986
56. Fowler, M.: Refactoring: improving the design of existing code. Addison-Wesley Professional (2018)
57. Johnston, B., Jones, A., Kruger, C.: Applied Unsupervised Learning with Python: Discover hidden patterns and relationships in unstructured data with Python. Packt Publishing (2019)
58. Jones, A.: Probability, Statistics and Other Frightening Stuff. Working Guides to Estimating & Forecasting. Taylor & Francis (2018)
59. Jurafsky, D., Martin, J.: 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)
60. Kamiya, T., Kusumoto, S., Inoue, K.: Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. IEEE Transactions on Software Engineering **28**(7), 654–670 (2002)
61. Kim, M., Gee, M., Loh, A., Rachatasumrit, N.: Ref-finder: a refactoring reconstruction tool based on logic query templates. In: Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering, pp. 371–372. ACM (2010)
62. Kim, M., Zimmermann, T., Nagappan, N.: An empirical study of refactoring challenges and benefits at microsoft. IEEE Transactions on Software Engineering **40**(7), 633–649 (2014). DOI 10.1109/TSE.2014.2318734
63. Lambiase, S., Cupito, A., Pecorelli, F., De Lucia, A., Palomba, F.: Just-in-time test smell detection and refactoring: The darts project. In: Proceedings of the 28th International Conference on Program Comprehension, ICPC '20, p. 441–445. Association for Computing Machinery, New York, NY, USA (2020). DOI 10.1145/3387904.3389296
64. Lane, H., Hapke, H., Howard, C.: Natural Language Processing in Action: Understanding, Analyzing, and Generating Text with Python. Manning Publications Company (2019)
65. Liu, H., Liu, Q., Liu, Y., Wang, Z.: Identifying renaming opportunities by expanding conducted rename refactorings. IEEE Transactions on Software Engineering **41**(9), 887–900 (2015)
66. Mazinanian, D., Tsantalis, N., Stein, R., Valenta, Z.: Jdeodorant: clone refactoring. In: Proceedings of the 38th international conference on software engineering companion, pp. 613–616 (2016)
67. Mens, T., Tourwe, T.: A survey of software refactoring. IEEE Transactions on Software Engineering **30**(2), 126–139 (2004). DOI 10.1109/TSE.2004.1265817

68. Mkaouer, W., Kessentini, M., Shaout, A., Koligheu, P., Bechikh, S., Deb, K., Ouni, A.: Many-objective software remodularization using nsga-iii. ACM Transactions on Software Engineering and Methodology (TOSEM) **24**(3), 1–45 (2015)
69. Moghadam, I.H., Cinnéide, M.Ó., Zarepour, F., Jahanmir, M.A.: Refdetect: A multi-language refactoring detection tool based on string alignment. IEEE Access (2021)
70. Moser, R., Sillitti, A., Abrahamsson, P., Succi, G.: Does refactoring improve reusability? In: International Conference on Software Reuse, pp. 287–297. Springer (2006)
71. Murphy-Hill, E., Parnin, C., Black, A.P.: How we refactor, and how we know it. IEEE Transactions on Software Engineering **38**(1), 5–18 (2012). DOI 10.1109/TSE.2011.41
72. Openja, M., Adams, B., Khomh, F.: Analysis of modern release engineering topics : – a large-scale study using stackoverflow –. In: 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 104–114 (2020). DOI 10.1109/ICSME46990.2020.00020
73. Ouni, A., Kessentini, M., Sahraoui, H., Inoue, K., Deb, K.: Multi-criteria code refactoring using search-based software engineering: An industrial case study. ACM Transactions on Software Engineering and Methodology (TOSEM) **25**(3), 23 (2016)
74. Palomba, F., Bavota, G., Di Penta, M., Oliveto, R., De Lucia, A., Poshyvanyk, D.: Detecting bad smells in source code using change history information. In: 2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 268–278. IEEE (2013)
75. Pantiuchina, J., Lanza, M., Bavota, G.: Improving code: The (mis) perception of quality metrics. In: 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 80–91. IEEE (2018)
76. Pantiuchina, J., Zampetti, F., Scalabrino, S., Piantadosi, V., Oliveto, R., Bavota, G., Penta, M.D.: Why developers refactor source code: A mining-based study. ACM Trans. Softw. Eng. Methodol. **29**(4) (2020). DOI 10.1145/3408302
77. Peruma, A.: A preliminary study of android refactorings. In: 2019 IEEE/ACM 6th International Conference on Mobile Software Engineering and Systems (MOBILESoft), pp. 148–149 (2019)
78. Peruma, A., Arnaoudova, V., Newman, C.D.: Ideal: An open-source identifier name appraisal tool. In: 2021 IEEE International Conference on Software Maintenance and Evolution (ICSME), ICSME '21 (2021)
79. Peruma, A., Hu, E., Chen, J., AlOmar, E., Mkaouer, M., Newman, C.D.: Using grammar patterns to interpret test method name evolution. In: 2021 2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC) (ICPC), pp. 335–346. IEEE Computer Society, Los Alamitos, CA, USA (2021). DOI 10.1109/ICPC52881.2021.00039. URL `https://doi-ieeecomputersociety-org.ezproxy.rit.edu/10.1109/ICPC52881.2021.00039`
80. Peruma, A., Mkaouer, M.W., Decker, M.J., Newman, C.D.: An empirical investigation of how and why developers rename identifiers. In: Proceedings of the 2nd International Workshop on Refactoring, IWoR 2018, p. 26–33. Association for Computing Machinery, New York, NY, USA (2018). DOI 10.1145/3242163.3242169. URL `https://doi.org/10.1145/3242163.3242169`
81. Peruma, A., Mkaouer, M.W., Decker, M.J., Newman, C.D.: Contextualizing rename decisions using refactorings, commit messages, and data types. Journal of Systems and Software **169**, 110704 (2020). DOI https://doi.org/10.1016/j.jss.2020.110704. URL `http://www.sciencedirect.com/science/article/pii/S0164121220301503`
82. Pinto, G.H., Kamei, F.: What programmers say about refactoring tools? an empirical investigation of stack overflow. In: Proceedings of the 2013 ACM Workshop on Workshop on Refactoring Tools, WRT '13, p. 33–36. Association for Computing Machinery, New York, NY, USA (2013). DOI 10.1145/2541348.2541357. URL `https://doi.org/10.1145/2541348.2541357`
83. Posnett, D., Hindle, A., Devanbu, P.: A simpler model of software readability. In: Proceedings of the 8th working conference on mining software repositories, pp. 73–82 (2011)
84. Roberts, D., Brant, J., Johnson, R.: A refactoring tool for smalltalk. Theory and Practice of Object systems **3**(4), 253–263 (1997)
85. Röder, M., Both, A., Hinneburg, A.: Exploring the space of topic coherence measures. In: Proceedings of the Eighth ACM International Conference on Web Search and Data Mining, WSDM '15, pp. 399–408. ACM, New York, NY, USA (2015). DOI 10.1145/2684822.2685324. URL `http://doi.acm.org.ezproxy.rit.edu/10.1145/2684822.2685324`
86. Rosen, C., Shihab, E.: What are mobile developers asking about? a large scale study using stack overflow. Empirical Software Engineering **21**(3), 1192–1223 (2016). DOI 10.1007/s10664-015-9379-3. URL `https://doi.org/10.1007/s10664-015-9379-3`
87. Roy, C.K., Cordy, J.R., Koschke, R.: Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. Science of computer programming **74**(7), 470–495 (2009)
88. Samarthyam, G., Suryanarayana, G., Sharma, T.: Refactoring for software architecture smells. In: Proceedings of the 1st International Workshop on Software Refactoring, IWoR 2016, p. 1–4. Association for Computing Machinery, New York, NY, USA (2016). DOI 10.1145/2975945.2975946
89. Scalabrino, S., Linares-Vásquez, M., Oliveto, R., Poshyvanyk, D.: A comprehensive model for code readability. Journal of Software: Evolution and Process **30**(6), e1958 (2018)
90. Sievert, C., Shirley, K.: Ldavis: A method for visualizing and interpreting topics. In: Proceedings of the workshop on interactive language learning, visualization, and interfaces, pp. 63–70 (2014)
91. Silva, D., Silva, J., Santos, G.J.D.S., Terra, R., Valente, M.T.O.: Refdiff 2.0: A multi-language refactoring detection tool. IEEE Transactions on Software Engineering (2020)
92. Silva, D., Tsantalis, N., Valente, M.T.: Why we refactor? confessions of github contributors. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, p. 858–870. Association for Computing Machinery, New York, NY, USA (2016). DOI 10.1145/2950290.2950305. URL `https://doi.org/10.1145/2950290.2950305`
93. Spectrum, I.: The top programming languages 2016. `https://spectrum.ieee.org/static/interactive-the-top-programming-languages-2016`. (Accessed on 06/05/2020)
94. Spectrum, I.: The top programming languages 2017. `https://spectrum.ieee.org/computing/software/the-2017-top-programming-languages`. (Accessed on 06/05/2020)
95. Spectrum, I.: The top programming languages 2018. `https://spectrum.ieee.org/static/interactive-the-top-programming-languages-2018`. (Accessed on 06/05/2020)

96. Spectrum, I.: The top programming languages 2019. https://spectrum.ieee.org/static/interactive-the-top-programming-languages-2019. (Accessed on 06/05/2020)
97. Taeger, D., Kuhnt, S.: Statistical Hypothesis Testing with SAS and R. Wiley (2014)
98. Tahir, A., Dietrich, J., Counsell, S., Licorish, S., Yamashita, A.: A large scale study on how developers discuss code smells and anti-pattern in stack exchange sites. Information and Software Technology p. 106333 (2020)
99. Tahir, A., Yamashita, A., Licorish, S., Dietrich, J., Counsell, S.: 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, p. 68–78. Association for Computing Machinery, New York, NY, USA (2018). DOI 10.1145/3210459.3210466. URL https://doi.org/10.1145/3210459.3210466
100. Tang, Y., Khatchadourian, R., Bagherzadeh, M., Ahmed, S.: Towards safe refactoring for intelligent parallelization of java 8 streams. In: Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, ICSE '18, p. 206–207. Association for Computing Machinery, New York, NY, USA (2018). DOI 10.1145/3183440.3195098. URL https://doi.org/10.1145/3183440.3195098
101. Tashakkori, A., Teddlie, C., Teddlie, C.: Mixed Methodology: Combining Qualitative and Quantitative Approaches. Applied Social Research Methods. SAGE Publications (1998)
102. Tian, F., Liang, P., Babar, M.A.: How developers discuss architecture smells? an exploratory study on stack overflow. In: 2019 IEEE International Conference on Software Architecture (ICSA), pp. 91–100 (2019)
103. Tsantalis, N., Mansouri, M., Eshkevari, L.M., Mazinanian, D., Dig, D.: Accurate and efficient refactoring detection in commit history. In: Proceedings of the 40th International Conference on Software Engineering. ACM (2018)
104. Villanes, I.K., Ascate, S.M., Gomes, J., Dias-Neto, A.C.: What are software engineers asking about android testing on stack overflow? In: Proceedings of the 31st Brazilian Symposium on Software Engineering, SBES'17, p. 104–113. Association for Computing Machinery, New York, NY, USA (2017). DOI 10.1145/3131151.3131157. URL https://doi.org/10.1145/3131151.3131157
105. Wang, S., Lo, D., Jiang, L.: An empirical study on developer interactions in stackoverflow. In: Proceedings of the 28th Annual ACM Symposium on Applied Computing, SAC '13, p. 1019–1024. Association for Computing Machinery, New York, NY, USA (2013). DOI 10.1145/2480362.2480557. URL https://doi.org/10.1145/2480362.2480557
106. Wilking, D., Kahn, U.F., Kowalewski, S.: An empirical evaluation of refactoring. e-Informatica **1**(1), 27–42 (2007)
107. Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., Wesslén, A.: Experimentation in software engineering. Springer Science & Business Media (2012)
108. Yang, X.L., Lo, D., Xia, X., Wan, Z.Y., Sun, J.L.: What security questions do developers ask? a large-scale study of stack overflow posts. Journal of Computer Science and Technology **31**(5), 910–924 (2016). DOI 10.1007/s11390-016-1672-0. URL https://doi.org/10.1007/s11390-016-1672-0