Industry Experiences with Large-Scale Refactoring

James Ivers
jivers@sei.cmu.edu
CMU Software Engineering Institute
Pittsburgh, PA, USA

Robert L. Nord
rn@sei.cmu.edu
CMU Software Engineering Institute
Pittsburgh, PA, USA

Ipek Ozkaya
ozkaya@sei.cmu.edu
CMU Software Engineering Institute
Pittsburgh, PA, USA

Chris Seifried
cgseifried@sei.cmu.edu
CMU Software Engineering Institute
Pittsburgh, PA, USA

Christopher S. Timperley
ctimperley@cmu.edu
Carnegie Mellon University
Pittsburgh, PA, USA

Marouane Kessentini
kessentini@oakland.edu
Oakland University
Rochester, MI, USA

ABSTRACT

Software refactoring plays an important role in software engineering. Developers often turn to refactoring when they want to restructure software to improve its quality without changing its external behavior. Small-scale (floss) refactoring is common in industry and is often performed by a single developer in short sessions, even though developers do much of this work manually instead of using refactoring tools. However, some refactoring efforts are much larger in scale, requiring entire teams and months or years of effort, and the role of tools in these efforts is not as well studied. In this paper, we report on a survey we conducted with developers to understand large-scale refactoring and its tool support needs. Our results from 107 industry developers demonstrate that projects commonly go through multiple large-scale refactorings, each of which requires considerable effort. Our study finds that developers use several categories of tools to support large-scale refactoring and rely more heavily on general-purpose tools like IDEs than on tools designed specifically to support refactoring. Tool support varies across the different activities, with some particularly challenging activities seeing little use of tools in practice. Furthermore, our analysis suggests significant impact is possible through advances in tool support for comprehension and testing, as well as through support for the needs of business stakeholders.

CCS CONCEPTS

• **Software and its engineering → Software evolution**; **Maintaining software**; **Software maintenance tools**; **Development frameworks and environments**.

KEYWORDS

refactoring, large-scale refactoring, refactoring tools, software automation, software evolution

ACM Reference Format:
James Ivers, Robert L. Nord, Ipek Ozkaya, Chris Seifried, Christopher S. Timperley, and Marouane Kessentini. 2022. Industry Experiences with Large-Scale Refactoring. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '22), November 14–18, 2022, Singapore, Singapore.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3540250.3558954

ESEC/FSE '22, November 14–18, 2022, Singapore, Singapore

ACM ISBN 978-1-4503-9413-0/22/11.
https://doi.org/10.1145/3540250.3558954

1 INTRODUCTION

Refactoring is defined as restructuring software to improve its quality without altering its external behavior [28]. The need to restructure software can come from such diverse goals as improving software quality, migrating to new platforms like cloud, containerizing software for DevOps, incorporating new technologies, or extracting capabilities for strategic reuse. Many of these scenarios involve broad changes to the system that cannot be accomplished through local code changes. This paper focuses on these larger refactoring efforts, which we refer to as **large-scale refactoring (LSR)**. Our experience working with multiple government and industry organizations on such refactoring efforts shows that such large-scale endeavors often represent months to years of effort.

Well-known refactoring types described by Martin Fowler (e.g., rename, move function, extract class) are frequently used by developers [8]. However, in industry, manual efforts dominate use of available tool support for refactoring [24–27, 37]. Integrated development environments (IDEs) like IntelliJ IDEA, Eclipse, VS Code, and Visual Studio all include features that change code to apply primitive refactoring types as directed by users. However, these tools have varying levels of acceptance by developers, even for their targeted local refactoring use cases. For example, in a study done with 328 Microsoft developers, Kim et al. found that 86% refactor manually, with minimal use of features that implement the refactoring types they intend to use [15].

These well-established refactoring types are building blocks for large-scale refactoring [15, 40, 41]. Prior studies show use of such refactorings to address evolution of APIs [7, 13, 39], design [3], and architecture [2, 3, 20, 23, 36, 42]. Broad changes inherent in large-scale refactoring, however, are often hindered by software complexity and require labor-intensive efforts to complete. Consequently, developers continue to desire more time to conduct refactoring activities [6], often combine refactoring with new feature development to gain approval to proceed [14], and seek more tool support while not trusting tools that are available as previous empirical studies reveal [15, 25]. While large-scale refactoring is commonly performed in industry, empirical data to guide priorities for improving tool support for large-scale refactoring does not exist.

To understand how developers engage with large-scale refactoring and how they use tools to support different activities involved,

we conducted a developer survey. Our results demonstrate that developers use several categories of tools beyond IDE-provided refactoring features to support large-scale refactoring. Tool support varies across the different activities that are involved in large-scale refactoring, with some particularly challenging activities seeing little use of tools in practice. While developers broadly agree that better tools are desired, they vary in the activities and degree of intelligence they want in tools.

The contributions of this study include the following:

- We share analysis results specifically focused on large-scale refactoring, demonstrate its prevalence with empirical data from industry, and position it as part of refactoring research and tool agendas. This contribution provides empirical data that challenges assumptions that research should focus on small-scale (floss) refactoring.
- We identify common reasons for deciding to perform or forgo large-scale refactoring. We also identify common consequences of forgoing such refactoring, adding a missing perspective to industry's motivation in engaging in large-scale refactoring.
- We identify which refactoring activities are most challenging, time consuming, and see the greatest and least use of tools. We also identify the categories of tools that are used today, further improving our understanding of gaps in refactoring tool support.
- We recommend comprehension and testing related features as high impact features based on our analysis of what industry developers report as missing and useful for large-scale refactoring tool support.
- Lastly, we share our data.

To the best of our understanding, this study is the first of its kind solely focused on understanding large-scale refactoring across multiple organizations.

2 BACKGROUND

Refactoring is a complex activity involving problem recognition, problem analysis, decision making, implementation, and evaluation [10].

Murphy-Hill and Black [25] introduced two different notions of refactoring; the need to continually tweak code while making other changes (floss refactoring) and infrequent, but focused changes to improve unhealthy code (root-canal refactoring). Floss and root-canal refactoring are primarily differentiated by the nature of changes made – floss refactoring intermingles refactoring with other changes, like feature development, while root-canal is almost entirely about refactoring. Root-canal refactoring is also typically described as correcting unhealthy code, emphasizing quality improvements rather than other motivations. Multiple empirical studies that use analysis of commit histories or IDE usage have found more evidence of floss refactoring than evidence of root-canal refactoring [22, 26, 35]. Murphy-Hill et al. [26] further suggest that "studies should focus on floss refactoring for the greatest generality."

Tools that implement refactorings in code are available for many popular programming languages through IDE context menu options that provide a list of available refactoring types from which developers choose, such as those included in IntelliJ IDEA, Eclipse, VS Code, and Visual Studio. According to a recent JetBrains survey of 1183 respondents, developers do in fact refactor their code every week or even almost daily and refactoring sessions often last an hour or longer. Despite this tool support, developers frequently refactor their code manually, often due to a lack of trust in what tools would do [9]. In addition, they seek more advice on topics like how to refactor their code to achieve objectives like code analyzabililty and readability than on how to use these tools as demonstrated by refactoring questions asked in Stack Overflow [29]. Furthermore, studies analyzing GitHub contributions reveal that refactorings are driven more often by changing requirements than by code smells [34]. There is clearly an aspect of refactoring that is broader in scope than the local code improvements that its original definition and floss refactoring recommendations implied.

Large-scale refactoring is restructuring of software, without introducing functionality, for the purpose of improving non-functional quality or changing architecture. Large-scale refactoring is a kind of root-canal refactoring in that both focus on structural improvements that are not intermingled with other changes. However, the challenges that arise as the scale and complexity of refactoring increases merit separate study. While floss refactoring often occurs over short bursts (e.g., minutes to hours), root-canal has a less clear scope, spanning anything from days to years of effort. For large-scale refactoring, we focus on efforts that require months to years of effort.

Large-scale refactoring involves either pervasive changes across a codebase or extensive changes to a substantial element of the system. Large-scale refactoring also often suggests a substantial commitment of resources, requiring management approval. One example is the need to partition legacy monoliths into smaller pieces to create separately deployable, scalable, and evolvable units. Another is restructuring interfaces and communication patterns to enable replacement of a legacy feature by an improved or less proprietary alternative.

Many examples of root-canal refactorings in the literature do not represent particularly large efforts that require management support and significant commitment of resources. Using this distinction in our survey, we captured data from multiple organizations for refactoring efforts that were estimated to require a mean of more than 1500 staff days of effort and that were often motivated by broader business concerns than quality improvement alone.

Architecture refactoring overlaps significantly with large-scale refactoring. Large-scale refactoring efforts typically involve architecture changes; however, not all rearchitecting efforts require large-scale efforts. Several research efforts are developing tools that recommend sequences of refactorings that accomplish architecture-scale changes [12, 21, 38], but these efforts have yet to be incorporated in commercial tools. Other work in architecture refactoring addresses code and architecture smell detection, which often focus narrowly on quality symptoms as indicators of opportunities for architecture-scale changes [31, 35]. Our study, in contrast, takes a broad perspective on the range of activities and supporting tools from the perspective of developers performing large-scale refactoring in industry.

Refactoring process studies are important to understand gaps in tool support. A recent study of how software developers make decisions proposed a decision-making framework for refactoring [19].

They found stages of decision-making that consist of a pain zone that triggers a decision to refactor, situation analysis, refactoring planning, refactoring implementation, and follow up to assess the effort. Factors that lead to decision making are influenced by scale. More recently, Haendler and Frysak [10] provided a theoretical perspective on applying concepts from decision-making research to deconstruct the refactoring process. They provide a more general interpretation of the software maintenance process [17] and different refactoring stages [19]: problem recognition, problem analysis, decision-making, implementation, evaluation. Furthermore, the model introduces a second dimension to account for the primary decisions in refactoring at management and operational levels: whether to refactor, what to refactor, how to refactor. The authors then group the many tools and techniques available for refactoring by the following characteristics: smell detection and refactoring recommendation tools, code-quality and design-critique tools, refactoring tools, technical debt management and analysis tools, automated regression testing frameworks, and documented knowledge on refactoring rules. Our survey also reveals tools used across these categories and confirms that the support is not ideal.

These studies commonly point out that the refactoring process consists of activities that span several decision-making stages as well as activities along the software development lifecycle. Abid et al. recently completed a literature survey spanning 30 years of refactoring research that emphasized a lifecycle view of refactoring [1]. In our survey, we build on these studies and focus on the following activities:

- Determining where changes are needed
- Choosing what changes to make
- Implementing the changes
- Generating new tests
- Migrating existing tests
- Validating refactored code (inspection, executing tests, etc.)
- Re-certifying refactored code (common to industry in regulated domains)
- Updating documentation

Through the rest of this paper, we use these activities to understand the prevalence of large-scale refactoring, its challenges, and gaps in existing tools that support these activities. There are other survey studies of refactoring, however, none focused on large-scale refactoring as an industry relevant problem affecting the longevity and maintainability of systems. Kim et al.'s [15] survey of developers at Microsoft in 2014 had the goal of understanding the benefits of refactoring and developer perceptions. Their conclusions included that the definition of refactoring in practice is broader than behavior-preserving program transformations and include system wide changes. In addition, they showed that developers need various types of refactoring support beyond the refactoring features provided by IDEs. Golubev et al.'s survey study focused specifically on IntelliJ users and their perceptions of trust of IntelliJ's refactoring features [9]. Our survey study is similar in its methodology to these other survey studies like Kim and Golubev; however, it differs in its motivation and establishes large-scale refactoring as a distinct refactoring category.

RQ1	• What were the business goals of the refactoring? • Have you ever wanted to perform a large-scale refactoring but were unable to? • What consequences, if any, did you observe from not performing the refactoring?
RQ2	• What tools, if any, did you use to assist your large-scale refactoring efforts? • To what extent do you use tools for the following activities?
RQ3	• What kind of automation, if available, would have most improved your large-scale refactoring? • What are the strengths and weakness of the tools you used to support large-scale refactoring?

Figure 1: A sample of our survey questions and their corresponding research question (RQ).

3 METHODOLOGY

Our goals in this study include assessing how developers perform large-scale refactoring and understanding the tools they use to support the process and their shortcomings. To achieve these goals, we ask the following research questions:

RQ1: How pervasive is large-scale refactoring in industry?

RQ2: How do developers use tools to aid their large-scale refactoring efforts?

RQ3: What tools and support, if any, do developers desire to aid their large-scale refactoring efforts?

In addition to assessing how common large-scale refactoring is, we also collect information related to its overarching business and technical goals, reasons why and why not to refactor, and risks and challenges associated with large-scale refactoring. Our other questions focus on refactoring process activities, examine the role of tools to support these activities, and elicit what kind of tools would better support these activities.

Survey Design. To answer our research questions, we performed an online survey of members of the software engineering community between November 2020 and February 2021. To ensure that we collected meaningful and informative results, we followed several survey design best practices by explicitly deriving survey questions from our research questions, conducting a series of iterative pilot surveys on a representative population of sample respondents, and refining our survey design until reaching saturation [4, 16]. A sample of our survey questions is given in Figure 1. We used a branching design to elicit separate experiences in which participants had performed large-scale refactoring and those in which they had been unable to do so. Those who had performed large-scale refactoring were presented questions related to the challenges, outcomes, and the extent to which tools supported the process. Those who were unable to do so answered questions as to why not and the consequences of not refactoring. 80 respondents completed the branch of questions for those who had performed large-scale refactoring, 61 respondents completed the branch for having been unable to do so, and 47 respondents completed both branches.

Recruitment. Our survey targeted an industry audience. We distributed our survey to members of the software engineering community via email (dlist: 7,700), LinkedIn (subscribers: 16,012), Twitter (followers: 5,383), research colleagues (for redistribution

Table 1: Demographics of our 107 survey participants in terms of their years of experience in the software industry.

Years of experience	#	%
Less than three years	6	6%
Between three and ten years	16	15%
Ten or more years	85	79%

to their industry collaborators), and company internal technical interest groups. A total of 107 participants took part in the survey. 74% of participants worked in industry and 96% of participants have worked as software engineers and/or software architects (both of which we refer to as developers). 79% of the participants had 10+ years of experience (Table 1). These demographics demonstrate that our participants represent a wealth of collective industry experience, which helps to increase the confidence in our findings.

Qualitative Analysis. To analyze responses to the qualitative parts of our survey, we used a descriptive coding approach [32]. Two authors tagged each response to open-ended survey questions with one or more labels, known as codes, describing the topics of that response. We then performed adjudication with a third author to resolve any disagreements and code mapping to collapse our codes into a consistent set of categories. Finally, we used axial coding to identify relationships between categories, and to identify a small number of overarching themes. Throughout this process, we performed continual analysis, comparison, and discussion of data until reaching thematic saturation (i.e., no new perspectives, dimensions, or relationships were identified).

These responses reflect respondents' perceptions and experiences of large-scale refactoring. We report frequency of our coding of this data only to demonstrate the prevalence of themes in our data, not to suggest generalized conclusions. To allow others to understand the logic behind our analysis process, we also provide sample quotes throughout the paper.

Study Artifacts. To promote further research and allow others to inspect and replicate our methodology and findings, we provide a detailed audit trail of our study artifacts, which include our survey questionnaire, recruitment materials, codebook, anonymized survey data, and the Jupyter notebook used to produce the figures in the paper. Our study artifacts are available at:
https://github.com/ArchitecturePractices/lsr_survey_artifacts.

4 LSR IS AN OPEN INDUSTRY CHALLENGE

RQ1 asked how pervasive large-scale refactoring is in industry. Common wisdom says that business priorities and natural system evolution drive the need to conduct large-scale changes in industrial software. However, we do not know whether developers resonate with the concept of large-scale refactoring and how frequently, if at all, they engage in conscious large-scale refactoring activities. We wanted to understand the business and technical triggers, as well as the challenges surrounding the decision to perform or forgo large-scale refactoring and their consequences from the perspective of industry developers.

4.1 Prevalence of LSR

Our findings confirm that large-scale refactoring is a major undertaking that can be performed on industry software multiple times in its lifetime. 82% of respondents had participated in large-scale refactoring at least once, with 61% reporting participating in large-scale refactoring more than once and 12% reporting engaging in such refactoring five or more times. These refactorings were performed on significantly large systems (34% were larger than 1M LOC and 38% ranged from 100K-1M LOC) and consumed significant resources, ranging from 2 days to 20,000 staff days as shown in Figure 2 and 86.4% of whom reported effort that falls in the months to years range.

Furthermore, 56% of systems on which respondents had performed large-scale refactoring had undergone large-scale refactoring multiple times (16% twice, 36% three to five times, and 5% more than five times). Half of respondents reported that they are still working on the same system on which they had performed large-scale refactoring, 42% of whom have worked on this system for more than five years. The release frequencies for these systems ranged from several times a month (25%) to several times a year (49%). These results confirm that industrial software systems go through major changes and support the conclusion that organizations do commonly conduct large-scale refactoring.

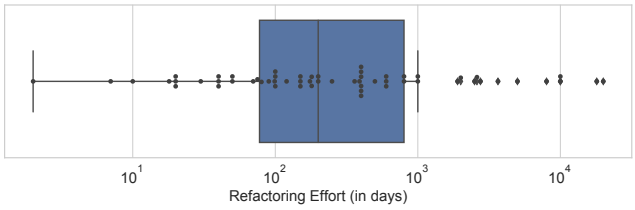

Figure 2: Estimated effort (in staff days) that teams required to complete their large-scale refactorings.

4.2 Reasons for LSR

Reducing cost of change and time to deliver were expressed as top business reasons to refactor by our respondents who both had the opportunity to refactor and wanted to refactor but could not (Figure 3). As for technical reasons for refactoring, improving understandability and migrating to a new architecture had similarly top occurrences (Figure 4).

Our analysis revealed an interesting relationship between the top business and technical reasons: 78% of those who reported that reducing cost of change was a business reason to refactor also reported improving code understandability as a technical reason to refactor. Among those who did undertake large-scale refactoring, roughly 70% reported both improving code understandability and migrating to a new architecture as top technical reasons to refactor. These results further demonstrate the relationship between the kind of architectural change that requires large-scale refactoring and the potential impact of such change on business goals.

The scope of work in large-scale refactoring is broader than local code improvements. The following survey response succinctly

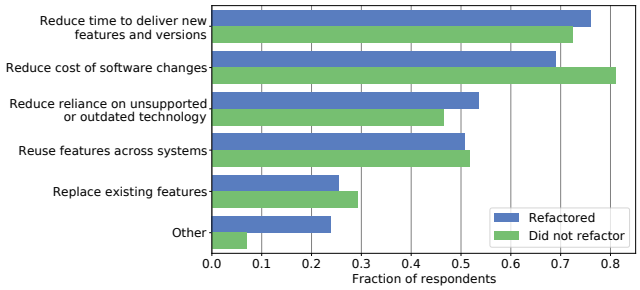

Figure 3: Business reasons for large-scale refactoring.

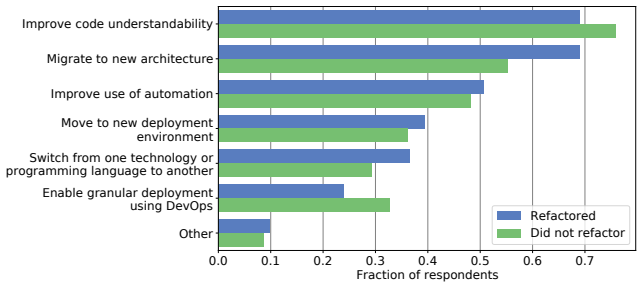

Figure 4: Technical reasons for large-scale refactoring.

summarizes how the decision making process differs in large-scale refactoring:

- *Agile development practices encourage continuous micro refactorings to make the code a little bit better all the time. ... Refactoring is just part of the job ... like a surgeon washing her hands. ...teams should be continuously refactoring in the small without the need for explicit investment or direction from the wider business. I think larger scale "refactoring" is different in that there is an opportunity cost to doing or not doing the work. It becomes a business decision as to where to invest.*

4.3 Forgoing LSR

Having established that industry systems undergo multiple large-scale refactorings, we looked at how often organizations had wanted to perform refactoring but had decided not to do so. 71% of respondents reported that there were occasions that they wanted to conduct large-scale refactoring, but had not done so. Sharma's study reported a similarly high portion of respondents (76%) identifying prioritization of features over refactoring as an obstacle to undertaking refactoring [33]. The reasons for deciding not to perform large-scale refactoring center around opportunity cost (new features were prioritized and anticipated cost was too high) as the most important reasons. 35% of respondents reported both as driving reasons, indicating that when resources are scarce, new features are commonly preferred over other investments. Interestingly, among the reasons to not refactor, only 6% of the participants indicated that the anticipated value of refactoring was too low (Figure 5).

Given business realities, these results are not surprising. When resource constraints (especially time and cost) force choices, new

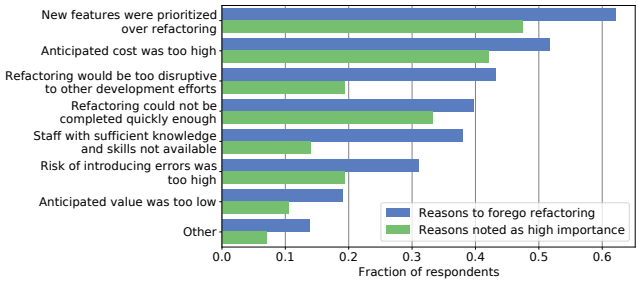

Figure 5: Reasons why organizations forgo large-scale refactoring.

Table 2: Consequences of forgoing large-scale refactoring, by fraction of respondents reporting each category.

Category	Description	%
Delivery	Slow feature delivery, inability to develop features	56%
Internal quality	Low productivity, duplicated code, non-bug design flaws	54%
External quality	Degraded user experience, bugs, performance issues	32%
Staffing	Low morale, increased onboarding time, difficulty hiring or retaining staff	22%

features are prioritized over refactoring. However, there are consequences to not performing needed refactoring, as our participants reported through open ended questions. When we analyzed these responses through a coding exercise (Table 2), we found that the most common long term consequences were related to inability to or slowing pace of delivering new features (56%). Instances of deteriorating internal (54%) and external (32%) quality were often accompanied by references to increasing operating or development costs, which are expected consequences of quality deterioration. 90% of respondents reported delivery and/or internal quality problems, both of which reflect slowing development velocity, as consequences of not refactoring.

The consequences that participants shared also clearly exemplify the need for large-scale refactoring.

- *We are stuck on outdated technologies. It is difficult to keep up with the "startup" companies that provide features that we are not able to create on the old tech stack.*
- *...modernization cycle was held back by 4 years....maintenance cost stayed high....cost to implement, deploy, and validate continue to increase.*
- *Feature delivery took longer as it required changes to multiple parts of the system.*

These consequences undermine the perceived opportunity to divert resources from refactoring to new features.

Finding 1: Large-scale refactoring is prevalent in industry. 82% of respondents had performed large-scale refactoring. Of the systems on which they had performed large-scale refactoring, 57% had undergone multiple large-scale refactorings. Furthermore, 71% of respondents had wanted to perform large-scale refactoring, but were unable to do so.

Finding 2: Large-scale refactorings are substantial efforts. The mean time to complete refactoring is estimated at more than 1500 staff days.

Finding 3: Improving cost of change and time to deliver are the top business reasons to refactor, while improving code understandability and migrating to a new architecture are the top technical reasons.

Finding 4: Forgoing large-scale refactoring slows delivery tempo. While prioritizing new features over refactoring is the most common reason for forgoing large-scale refactoring, 56% of respondents report the inability to or slowing pace of delivering features as a consequence of forgoing large-scale refactoring.

5 INADEQUATE TOOL SUPPORT FOR LSR

Refactoring has been a familiar concept to developers for decades [8], but adoption of tools to support refactoring remains less common [15, 26]. While studies have focused more on support for floss refactoring, Kim et al.'s interviews included a team that had performed system-wide refactoring on a very large system [15]. Their analysis indicates that refactoring at this scale involves far more than applying refactorings for local changes. Instead, they observed that system-wide refactoring involved understanding the system, performing dependency analysis, creating a desired architecture structure, performing multiple gate checks, educating other developers, and developing custom refactoring tools. In our RQ2, we sought to understand what kinds of tools were used in large-scale refactoring and whether they differ from those used in other refactoring. We also investigated the different activities involved in large-scale refactoring and how those tools support those activities.

5.1 Tools Used

We used two open ended questions to collect a list of tools that respondents used for refactoring at any scale and for large-scale refactoring. We used coding to categorize each tool into one of the categories shown in Figure 6, which contrasts the fraction of respondents using at least one tool in each category for refactoring at any scale with that for large-scale refactoring. There is little difference between the fraction of respondents using tools for large-scale refactoring and refactoring at any scale for most tool categories. The exceptions are IDEs and text editors (greater use in refactoring at any scale), testing tools (greater use in large-scale), and other tools (much greater use in large-scale). The other tools category includes custom scripts and tools on which custom tools were likely built (e.g., static code analyzers and abstract syntax trees).

The most commonly used category of tool is the IDE; more than half of all respondents reported using IDEs for refactoring (68.7% for any scale and 54.3% for large-scale). In contrast, fewer than 10% of respondents reported using tools that are designed specifically for refactoring like ReSharper and JDeodorant (8.4% for any scale and 4.3% for large-scale) or called out refactoring features of IDEs (6% for any scale and 6.5% for large-scale). The portion of tools falling into the other category was substantially higher for large-scale refactoring (50%) than for refactoring at any scale (18%).

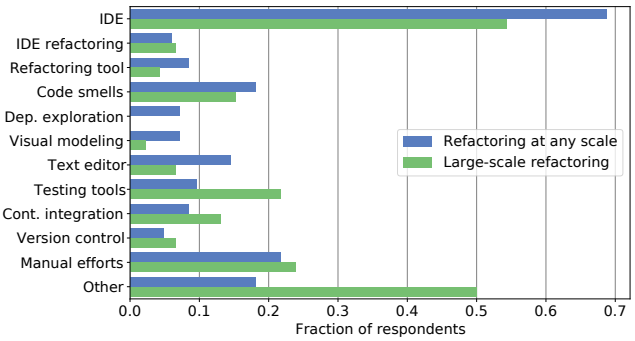

Figure 6: Categories of tools used to support refactoring.

5.2 Refactoring Activities

We next looked at the work that respondents perform as part of large-scale refactoring activities. We listed the refactoring activities found in Figure 7 and asked respondents to report how much time they spent in each, how challenging they found each, and the extent to which they used tools for each. Figure 7 shows the fraction of respondents reporting each activity in the positive for each question (i.e., most time spent, most challenging, and extensive use of tools). The top three activities in terms of taking the most time, being the most challenging, and making the greatest use of tools all come from these four activities: (1) determining where changes are needed, (2) choosing what changes to make, (3) implementing changes, and (4) validating refactored code.

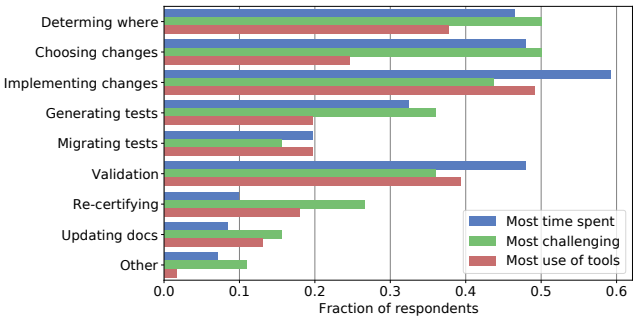

Figure 7: Refactoring activities that take the most time, are most challenging, and make the most use of tools.

Respondents commonly reported choosing what change to make as most time consuming (48%) and most challenging (50%), but only 25% reported it as making extensive use of tools. In fact, looking at the negative responses, 58% reported this activity as making the least use of tools. The activity for which respondents reported least use of tools was updating documentation (75%), which was also commonly noted as taking the least time (63%) and being least challenging (61%).

5.3 Tool Effectiveness

Ideally, there is a relation among how much tools are used for an activity, how much time it takes, and how challenging it is. Highly effective tools can dramatically reduce the time spent and the perceived challenge. Activities that remain highly challenging and time consuming can suggest shortcomings in or under-use of tools. We compared the tools that respondents used for large-scale refactoring (Figure 6) with the time spent, challenge, and extent of tool use for refactoring activities (Figure 7). We also applied our judgment regarding the degree of support each category of tool provides (based on the specific tools listed by respondents) to each activity as additional context.

Respondents reported that determining where changes are needed and implementing changes as highly challenging activities (50% and 43.8%) for which they extensively use tools (37.7% and 49.2%), and yet both take significant time (46% and 59%). Of note, respondents report relatively little use of tools that are specifically designed to support these tasks. Refactoring, code smell analysis, and dependency exploration tools better support determining where changes are needed, but are used by only 4.3%, 15.2%, and 0% of respondents. Refactoring tools and IDE refactoring features better support implementing changes, but are used by only 4.3% and 6.5% of respondents. Both indicate that while respondents use tools extensively for two of the three most challenging activities, they rely more heavily on general purpose tools like IDEs and manual effort than on tools specifically designed for refactoring.

Finding 5: Few respondents (less than 10%) indicate use of tools specifically designed for refactoring.

Finding 6: 50% of respondents performing large-scale refactoring report use of other tools, which are dominated by custom tools, scripts, and packages on which they build their own tools.

Finding 7: Choosing which changes to make is one of the most challenging and time consuming activities, while also one of the activities for which developers make the least use of tools.

Finding 8: Large-scale refactoring involves multiple activities that lack adequate tool support, including activities that see little use of tools and activities that take significant time despite extensive use of tools.

6 FOUNDATIONAL FEATURES ARE NEEDED

To understand what kinds of tools developers need and request for large-scale refactoring, we looked in RQ3 at the challenges respondents faced during their refactoring, the strengths and weaknesses of current tools, and how respondents directly answered the question.

6.1 Activity Challenges

After asking respondents to rate how challenging each refactoring activity was, we asked them through an open response question what made their most challenging activities challenging. Table 3 shows our coding of these responses. Unsurprisingly, the most common challenge is the poor quality of the software being refactored, a challenge that refactoring exercises inherit as a starting point.

Table 3: What made large-scale refactoring challenging, by fraction of respondents reporting each category.

Category	Description	%
Code Quality	Poor quality of code being refactored, excessive dependencies that complicate changes	34%
Comprehension	Difficulties in understanding code structure, flow, and possible side-effects	26%
Tests	Lack of tests to ensure behavior	19%
Communication	Need to persuade management and teammates, gaining user trust	19%
Scoping	Managing expectations, deciding how much refactoring to do	15%
Documentation	Poor documentation, unclear intent	13%
Techniques	Lack of well-defined refactoring techniques	11%
Decision Criteria	Choosing the right changes	6%

The second most common challenge is the difficulty in understanding code and the implications of a change. One respondent emphasized this as *The hardest part was gaining a conceptual grasp of the overall code structure, and code flow, and understanding how one basic change – no matter how simple it appeared on the surface – might create consequences throughout the system.* While this challenge is more dependent on the tools and processes used for refactoring, the starting quality of code can exacerbate it. Half of respondents reporting code comprehension as a challenge also reported poor code quality or lack of documentation as a challenge. A need for code comprehension often stems from inheriting code written by someone else. Most respondents reported that the software on which they had performed large-scale refactoring was relatively old when they started working on it (for 27% it was already 5-10 years old and for 25% it was already more than 10 years old).

Respondents reported challenges that closely relate to code artifacts more than twice as often (code quality and comprehension at 34% and 26%) as challenges that relate to making decisions (scoping refactoring and decision criteria at 15% and 6%).

6.2 Assessment of Current Tools

We next looked at participant responses to a question on the strengths and weaknesses of the tools that they currently use. Table 4 shows our coding of these responses. Less than half of respondents provided any strengths, while only three respondents provided only strengths. The top categories for reported strengths were modification (automation of changes at 16%) and planning what to refactor (identifying opportunities for refactoring at 12%). The top categories for reported weaknesses were usability (learning curve and poor interfaces for tasks at 33%) and modification (lack of control over or unacceptable results from automated refactoring at 21%). This corroborates a finding of Pinto and Kamei's study, which identified usability as a key barrier to adoption of refactoring tools [30].

Several responses directly contrasted available refactoring support for small-scale changes with needs for large-scale refactoring. Examples include:

- *The tools I use don't offer any guides or hints related to large-scale refactoring. Their analysis features usually present only*

Table 4: Strengths and weaknesses of tools respondents use, by fraction of respondents reporting each category.

Category	Strengths	Weaknesses
Usability	7%	33%
Modification	16%	21%
Planning what to refactor	12%	19%
Analysis	9%	16%
Large-scale refactoring	5%	16%
Comprehension	2%	16%
Testing	2%	5%
Planning how to refactor	0%	5%
Scoping refactoring	0%	5%

low level code smells that often don't offer a considerable improvement in the quality of the software.

- *They address refactoring efforts at a component level. They don't address end to end scenarios and analysing dynamics. Today's tools I have used provide quite a lot indicators for increasing complexity and structure loss, but these are not enough to make large scale decision with reducing these effects leading to system failure.*
- *The tools I've got are too focused on munging text, or on refactoring that is syntactically simple enough that I don't really need help with it (maybe it saves time on typing, but typing time isn't the problem).*

6.3 Interest in Intelligent Tools

The activity challenges (Table 3) and the weaknesses of tools used (Table 4) point to room for improvement. We asked participants what kind of tools would have most improved their experience. Table 5 shows our coding of these open ended responses. Fewer respondents expressed interest in intelligent tools that make decisions for them than in foundational tools that act as directed by a developer, like performing requested analyses, making specified changes, and confirming the results of changes.

The three most common categories focused directly on the code being refactored. Testing focused on testing automation (46%), modification focused on automating code changes (26%), and analysis focused on understanding the code (23%). In contrast, categories that included recommending actions were much less common: planning what to refactor (recommending where changes are needed at 9%) and planning how to refactor (recommending specific changes at 6%). Pinto and Kamei's analysis of Stack Overflow questions on refactoring identified generating refactoring recommendations as a desirable feature at a similarly low number (13%) [30].

This preference aligns with Table 3, which summarizes what made refactoring challenging. Challenges with code comprehension and tests align with top requests. Decision criteria was the least common challenge, aligning with the lack of requests for intelligent tools that recommend changes.

Notably, this preference is somewhat at odds with where respondents report spending the most time in Figure 7. Two of the four activities on which they spend the most time (implementing changes and validation) align with two of the top three requests

Table 5: What kinds of tools would improve large-scale refactoring, by fraction of respondents reporting each category.

Category	%
Testing	46%
Modification	26%
Analysis	23%
Comprehension	17%
Planning what to refactor	9%
Build Automation	9%
Planning how to refactor	6%

(testing and modification). The other two activities on which they spend the most time (determining where changes are needed and choosing the changes to make) align with two of the least common requests (planning what and how to refactor).

While some respondents expressed clear skepticism of intelligent refactoring tools (e.g., *I'm still highly skeptical that a tool that can effectively automatically suggest a collection of refactoring that would solve a specific problem can be written.*), 73% of respondents replied that a tool that automatically suggests a collection of refactorings that would solve a problem that they specified would be useful. Regardless of any skepticism, responses to the question of what tools would help reflected a genuine, if sometimes plaintive need for help (e.g., *Any at all* and *Almost anything :-)*).

Finding 9: The most commonly reported refactoring challenge is the starting quality of software. This challenge is closely followed by the difficulty in comprehending that software.

Finding 10: Testing is the most common category of tools that respondents believe would improve large-scale refactoring.

Finding 11: Of the tools respondents use today, the most common strengths reported are in automating changes and the most common weaknesses are in usability.

Finding 12: Fewer respondents expressed interest in intelligent tools that make decisions for them than in foundational tools that act as directed by a developer.

7 DISCUSSION

Refactoring is an active area of research with an emphasis on tool support and particular strengths in detecting refactoring opportunities that improve code quality, safe implementation of low-level refactorings in code, and use of search-based techniques to generate refactoring recommendations [1]. Our survey results, despite representing only a sampling of the challenges and gaps in large-scale refactoring in industry, demonstrate that there is room for improvement in tools determining where to make changes and implementing those changes. However, our analysis of the open responses also points to other areas in which improvements would benefit industry teams executing large-scale refactorings – comprehension and testing. Furthermore, researchers and tool vendors have an opportunity to improve how they integrate their contributions into the broader refactoring lifecycle by addressing the needs of stakeholders beyond developers.

7.1 Comprehension Recommendations

In our survey, many responses across questions reinforce the importance of comprehension in large-scale refactoring. We defined the comprehension coding category to capture responses that focus on making sense of a codebase, including understanding existing code structure, design, and requirements. Many responses converge on a scenario that is anecdotally common in industry and for which comprehension is particularly crucial – documentation is missing or out-of-date, and so teams must discover how a large body of code is structured and which design strategies were employed (and why) prior to embarking on large-scale changes.

Our recommendations related to comprehension are based on analysis of and connections among responses to several questions. Determining where to make changes and choosing what changes to make, both activities that rely on first comprehending the current state of software, were noted as being the the most challenging activities by respondents (Figure 7). The top challenges in performing large-scale refactoring (Table 3) involved the prominence of code with excessive dependencies and the difficulty in understanding existing code structure. Additionally, the second most common category of tools that respondents directly asked for was a capability to help analyze code and the impact of possible changes (Table 5). The ability to comprehend existing code is challenging to respondents, critical to their most challenging activities, and an area in which help is demonstrably wanted.

The following responses to the question "What kind of automation, if available, would have most improved your large-scale refactoring?" illustrate tool gaps related to comprehension and point in constructive directions:

- *Configurable combination of data-flow, and control-flow analysis queries, pipe-lined with each other, and ability to create meaningful executable partitions of code*
- *static code analyzer on an architectural level to determine the efforts and tracking refactoring activities over time*
- *Anything that could have given me a conceptual overview of how the components of the system fit together, and influenced one another. Code diagrams or visualizations would have been most helpful. For instance, will this function behave differently depending on the value of some given global variable?*

When performing large-scale refactoring, the existing code is ground truth. However, when dealing with changes that span significant portions of a system or affect tens or hundreds of thousands of lines of code, developers benefit from working at a higher level of abstraction. Code-level analyses can produce overwhelming volumes of data, only some of which is pertinent to refactoring. Architecture and design models allow developers to focus on essential, strategic decisions without inclusion of all the details found in code.

We recommend integration of refactoring research with advances in research from related specialties like software architecture and reverse engineering to improve tool support for comprehension in the context of large-scale refactoring. Respondents want a way to comprehend their starting point and plan a path forward at a higher level than code. The architecture community long ago converged on the importance of thinking about architecture from multiple perspectives called views [5, 11, 18], each addressing specific concerns. However, research and tool support in this area primarily addresses views that focus on code structure (e.g., UML class diagrams) rather than views that convey run-time semantics like concurrency and protocols of interaction. Specific advances that would aid comprehension include:

- The ability to generate run-time architecture views from code.
- The ability to enrich dependency views with analyses of mechanisms that do not derive directly from programming languages, such as network-based communication, dependency injection, and information exchange via repositories.
- Extension of these capabilities across a broader range of programming languages (e.g., older languages that industry continues to rely on).

7.2 Testing Recommendations

Respondents likewise emphasized testing challenges through responses to several questions. Again, many responses indicated an anecdotally common scenario that significantly complicates the challenges of large-scale refactoring – insufficient tests were in place prior to beginning refactoring. From one response: *code coverage was quite low, so we worked with fear of breaking things (which we did in the end) and that made us overthink and double-triple check.*

Testing was the most common category of tools that respondents directly requested (Table 5). A lack of tests to ensure behavior was referenced by 19% of respondents as a challenge to their large-scale refactoring experiences (Table 3). Respondents reported little use of tools in generating or migrating tests (Figure 7). While validation activities did see greater use of tools, it was still among the most time consuming activities (Figure 7).

The following responses are illustrative of testing concerns:

- *cost of migrating tests matched cost of refactoring code*
- *certifying that no functionality was broken given lack of tests*
- *Cannot easily generate tests to ensure behaviour remains unchanged*
- *Better test coverage before we start. Would have saved the time we need to spend on the basic test suite to make sure we did not break anything.*

Testing is a critical activity when performing any refactoring as it can provide confidence that a refactoring has not altered behavior. Ideally, adequate tests exist prior to performing any refactoring, but this is not always the case, as evidenced by our results. Testing takes on greater significance in large-scale refactoring as the scale amplifies the time required to create missing tests, the difficulty in comprehending interactions among large segments of often messy code, and the importance of reassuring diverse stakeholders (e.g., managers) that refactoring is proceeding successfully and continued resources are warranted.

As with comprehension, research in related specialties like test generation is directly applicable to large-scale refactoring and their successful integration and refinement can improve tool support. Specific advances that would aid testing include:

- The ability to generate tests that specifically support refactoring changes. This could include scoping baseline test generation to the smallest collection of code that can be affected by specific proposed refactoring operations.

- The ability to generate consistent before and after tests for proposed refactoring operations, with transparent relations to improve developer confidence.
- The ability to generate non-functional tests to understand architectural implications of large-scale changes.

7.3 Stakeholder Recommendations

The large-scale refactorings reported in this survey required allocation and management of significant resources. While such efforts rely heavily on technical stakeholders like developers and architects, they also rely on on business stakeholders like project managers, executives with funding approval, and team leads of related systems.

Business stakeholders focus on concerns like cost, schedule, and competitive advantage that affect whether large-scale refactoring will be funded (Figure 3 and Figure 5). Their involvement led to inclusion of tools in our responses that are more often mentioned in project management circles than refactoring research, such as wikis and issue trackers.

We recommend research and tool support that address the needs of business stakeholders. The scope of large-scale-refactoring sharpens the need to keep track of potentially widespread or dependent series of changes that are more complex to manage across time and teams than is typical of floss refactoring. Specific advances needed include:

- Empirically validated resource estimation models for large-scale refactoring.
- Tools that track progress across the broad scope of changes needed, including identification of stable checkpoints for incremental review, test, and deployment.
- Refactoring strategies that favor incremental demonstration of value to retain management support.

Such advances would enhance planning and oversight of the complex, expensive efforts that large-scale refactoring entails. Furthermore, they would improve researchers' and tool vendors' understanding of the anticipated benefits of their work.

Our respondents were primarily technical stakeholders (96% being software engineers and/or software architects), but the challenges they reported spanned business and technical concerns. Their feature requests were oriented towards foundational capabilities whose use developers would direct themselves rather than than towards intelligent tools that make decisions on their behalf. While some appetite for intelligent tools exists, there is evidence of a lack of trust in automated decisions.

We recommend delivery of research and tools that provide foundational capabilities for technical stakeholders, such as our comprehension and testing recommendations. This provides industry with incremental delivery of value and opportunities for developers to innovate with different compositions of capabilities. For intelligent tools, providing transparency and control (e.g., confirmations before changes) are essential to building trust and gaining feedback on how to improve the capabilities that are delivered. It is likely that observable progress in foundational capabilities will help to overcome some barriers that our and other previous studies have identified and aid transition to practice of more intelligent tool support for refactoring in industry.

7.4 Threats to Validity

Our threats to validity include the following:

Internal Validity. Our analysis of survey responses represents a potential threat to internal validity. To mitigate this threat and to ensure the reliability of our qualitative findings, we implemented and consistently adhered to established guidelines and best practices for conducting qualitative research, including comprehensive data use, constant comparison, the use of tables, and refinement of codes through adjudication and investigator triangulation.

External Validity. Our findings are based on the data we collected from 107 survey respondents. We do not make a generalizability claim, but position our findings as observations supported by our data and research literature.

Conclusion Validity. We distributed our survey to a broad audience to collect the most relevant data for our goals. To ensure that we asked the right questions and to avoid introducing our own biases into the wording and selection of questions, we conducted a series of iterative pilots to identify and address shortcomings in the survey design. Furthermore, we included several open-ended questions to allow participants to share their views and experiences.

8 CONCLUSION

To understand the prevalence, challenges, and tool support for large-scale refactoring we conducted a survey with industry developers. Our analysis of data from 107 respondents, 79% of whom reported having at least 10 years of experience, confirm that large-scale refactoring is not an unusual occurrence. Industry systems undergo multiple large-scale refactorings over their lifetimes and the magnitude of effort involved in each is considerable. Refactoring tools designed to support floss refactoring efforts are not enough to address the breadth of activities that developers consider a part of large-scale refactoring, and developers encounter a wide range of challenges despite using many different kinds of tools. Our study demonstrates a clear need for better tools and an opportunity for researchers to make a difference in industry.

Ironically, despite identifying many challenges and weaknesses in today's refactoring tools, 80% of respondents report having achieved their large-scale refactoring goals. Given the many negative consequences of forgoing refactoring, developers have little choice but to refactor their software despite the significant amount of effort that our results show are required. Researchers and tool developers have an opportunity to make significant industry impact by eliminating even a fraction of the time spent on large-scale refactoring activities.

ACKNOWLEDGMENTS

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. References herein to any specific commercial product, process, or service by trade name, trade mark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by Carnegie Mellon University or its Software Engineering Institute. DM22-0444

REFERENCES

[1] Chaima Abid, Vahid Alizadeh, Marouane Kessentini, Thiago do Nascimento Ferreira Ferreira, and Danny Dig. 2020. 30 Years of Software Refactoring Research: A Systematic Literature Review. arXiv:2007.02194 [cs.SE]

[2] Davide Arcelli, Vittorio Cortellessa, Mattia D'Emidio, and Daniele Di Pompeo. 2018. EASIER: An Evolutionary Approach for Multi-objective Software ArchItecturE Refactoring. In *2018 IEEE International Conference on Software Architecture (ICSA)*. IEEE, 105–10509.

[3] Gabriele Bavota, Andrea De Lucia, Andrian Marcus, and Rocco Oliveto. 2014. Recommending Refactoring Operations in Large Software Systems. In *Recommendation Systems in Software Engineering*. Springer Berlin Heidelberg, Berlin, Heidelberg, 387–419.

[4] Marcus Ciolkowski, Oliver Laitenberger, Sira Vegas, and Stefan Biffl. 2003. *Practical Experiences in the Design and Conduct of Surveys in Empirical Software Engineering*. Springer Berlin Heidelberg, Berlin, Heidelberg, 104–128.

[5] Paul Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Robert Nord, and Judith Stafford. 2003. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley Professional.

[6] Danny Dig, William G. Griswold, Emerson R. Murphy-Hill, and Max Schäfer. 2014. The Future of Refactoring (Dagstuhl Seminar 14211). *Dagstuhl Reports* 4, 5 (2014), 40–67.

[7] Danny Dig and Ralph Johnson. 2005. The Role of Refactorings in API Evolution. In *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM '05)*. IEEE Computer Society, USA, 389–398.

[8] Martin Fowler. 1999. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[9] Yaroslav Golubev, Zarina Kurbatova, Eman Abdullah AlOmar, Timofey Bryksin, and Mohamed Wiem Mkaouer. 2021. One Thousand and One Stories: A Large-Scale Survey of Software Refactoring. *CoRR* abs/2107.07357 (2021). arXiv:2107.07357 https://arxiv.org/abs/2107.07357

[10] Thorsten Haendler. and Josef Frysak. 2018. Deconstructing the Refactoring Process from a Problem-solving and Decision-making Perspective. In *Proceedings of the 13th International Conference on Software Technologies - ICSOFT,*. INSTICC, SciTePress, 363–372.

[11] 2000. IEEE Recommended Practice for Architectural Description for Software-Intensive Systems. *IEEE Std 1471-2000* (2000), 1–30.

[12] James Ivers, Chris Seifried, and Ipek Ozkaya. 2022. Untangling the Knot: Enabling Architecture Evolution with Search-Based Refactoring. In *2022 IEEE 19th International Conference on Software Architecture (ICSA)*. 101–111.

[13] Miryung Kim, Dongxiang Cai, and Sunghun Kim. 2011. An Empirical Investigation into the Role of API-Level Refactorings during Software Evolution. In *Proceedings of the 33rd International Conference on Software Engineering* (Waikiki, Honolulu, HI, USA) *(ICSE '11)*. Association for Computing Machinery, New York, NY, USA, 151–160.

[14] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. 2012. A field study of refactoring challenges and benefits. In *20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-20), SIGSOFT/FSE'12, Cary, NC, USA - November 11 - 16, 2012*, Will Tracz, Martin P. Robillard, and Tevfik Bultan (Eds.). ACM, 50.

[15] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. 2014. An empirical study of refactoring challenges and benefits at Microsoft. *IEEE Transactions on Software Engineering* 40, 7 (2014), 633–649.

[16] Barbara A. Kitchenham and Shari L. Pfleeger. 1995 and 1996. Principles of Survey Research: Parts 1 – 6. *Software Engineering Notes* (1995 and 1996).

[17] Barbara A. Kitchenham, Guilherme H. Travassos, Anneliese von Mayrhauser, Frank Niessink, Norman F. Schneidewind, Janice Singer, Shingo Takada, Risto Vehvilainen, and Hongji Yang. 1999. Towards an Ontology of Software Maintenance. *Journal of Software Maintenance* 11, 6 (Nov. 1999), 365–389.

[18] P.B. Kruchten. 1995. The 4+1 View Model of architecture. *IEEE Software* 12, 6 (1995), 42–50.

[19] Marko Leppänen, Samuel Lahtinen, Kati Kuusinen, Simo Mäkinen, Tomi Männistö, Juha Itkonen, Jesse Yli-Huumo, and Timo Lehtonen. 2015. Decision-making framework for refactoring. In *2015 IEEE 7th International Workshop on Managing Technical Debt (MTD)*. 61–68.

[20] Yun Lin, Xin Peng, Yuanfang Cai, Danny Dig, Diwen Zheng, and Wenyun Zhao. 2016. Interactive and guided architectural refactoring with search-based recommendation. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '16)*. ACM, Seattle, US, 535–546.

[21] Yun Lin, Xin Peng, Yuanfang Cai, Danny Dig, Diwen Zheng, and Wenyun Zhao. 2016. Interactive and guided architectural refactoring with search-based recommendation. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 535–546.

[22] Hui Liu, Yuan Gao, and Zhendong Niu. 2012. An Initial Study on Refactoring Tactics. In *2012 IEEE 36th Annual Computer Software and Applications Conference*. 213–218.

[23] Mohamed Wiem Mkaouer, Marouane Kessentini, Slim Bechikh, Mel Ó Cinnéide, and Kalyanmoy Deb. 2016. On the use of many quality attributes for software refactoring: a many-objective search-based software engineering approach. *Empirical Software Engineering* 21, 6 (2016), 2503–2545.

[24] G.C. Murphy, M. Kersten, and L. Findlater. 2006. How are Java software developers using the Eclipse IDE? *IEEE Software* 23, 4 (2006), 76–83.

[25] Emerson Murphy-Hill and Andrew P. Black. 2008. Refactoring Tools: Fitness for Purpose. *IEEE Software* 25, 5 (2008), 38–44.

[26] Emerson Murphy-Hill, Chris Parnin, and Andreaw P. Black. 2012. How We Refactor, and How We Know It. *IEEE Transactions on Software Engineering* 38, 1 (2012), 5–18.

[27] Emerson R Murphy-Hill and Andrew P Black. 2007. Why don't people use refactoring tools?. In *WRT*. 60–61.

[28] William F Opdyke. 1992. Refactoring object-oriented frameworks. (1992).

[29] Anthony Peruma, Steven Simmons, Eman Abdullah AlOmar, Christian D. Newman, Mohamed Wiem Mkaouer, and Ali Ouni. 2022. How do i refactor this? An empirical study on refactoring trends and topics in Stack Overflow. *Empir. Softw. Eng.* 27, 1 (2022), 11.

[30] Gustavo H. Pinto and Fernando Kamei. 2013. What Programmers Say about Refactoring Tools? An Empirical Investigation of Stack Overflow. In *Proceedings of the 2013 ACM Workshop on Workshop on Refactoring Tools* (Indianapolis, Indiana, USA) *(WRT '13)*. Association for Computing Machinery, New York, NY, USA, 33–36.

[31] Luca Rizzi, Francesca Arcelli Fontana, and Riccardo Roveda. 2018. Support for Architectural Smell Refactoring. In *Proceedings of the 2nd International Workshop on Refactoring* (Montpellier, France) *(IWoR 2018)*. Association for Computing Machinery, New York, NY, USA, 7–10.

[32] Johnny M Saldana. 2015. *The Coding Manual for Qualitative Researchers. 3rd ed.* SAGE Publication.

[33] Tushar Sharma, Girish Suryanarayana, and Ganesh Samarthyam. 2015. Challenges to and Solutions for Refactoring Adoption: An Industrial Perspective. *IEEE Software* 32, 6 (2015), 44–51.

[34] Danilo Silva, Nikolaos Tsantalis, and Marco Tulio Valente. 2016. Why we refactor? confessions of GitHub contributors. *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Nov 2016).

[35] Leonardo Sousa, Willian Oizumi, Alessandro Garcia, Anderson Oliveira, Diego Cedrim, and Carlos Lucena. 2020. When Are Smells Indicators of Architectural Refactoring Opportunities: A Study of 50 Software Projects. In *Proceedings of the 28th International Conference on Program Comprehension* (Seoul, Republic of Korea) *(ICPC '20)*. Association for Computing Machinery, New York, NY, USA, 354–365.

[36] Ricardo Terra, Marco Tulio Valente, Krzysztof Czarnecki, and Roberto S Bigonha. 2012. Recommending refactorings to reverse software architecture erosion. In *2012 16th European Conference on Software Maintenance and Reengineering*. IEEE, 335–340.

[37] Mohsen Vakilian, Nicholas Chen, Stas Negara, Balaji Ambresh Rajkumar, Brian P. Bailey, and Ralph E. Johnson. 2012. Use, disuse, and misuse of automated refactorings. In *2012 34th International Conference on Software Engineering (ICSE)*. 233–243.

[38] Hanzhang Wang, Marouane Kessentini, and Ali Ouni. 2021. Interactive Refactoring of Web Service Interfaces Using Computational Search. *IEEE Trans. Serv. Comput.* 14, 1 (2021), 179–192.

[39] Peter Weißgerber and Stephan Diehl. 2006. Are Refactorings Less Error-Prone than Other Changes?. In *Proceedings of the 2006 International Workshop on Mining Software Repositories* (Shanghai, China) *(MSR '06)*. Association for Computing Machinery, New York, NY, USA, 112–118.

[40] Hyrum Wright. 2019. Lessons Learned from Large-Scale Refactoring. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 366–366.

[41] Hyrum K. Wright, Daniel Jasper, Manuel Klimek, Chandler Carruth, and Zhanyong Wan. 2013. Large-Scale Automated Refactoring Using ClangMR. In *2013 IEEE International Conference on Software Maintenance*. 548–551.

[42] Olaf Zimmermann. 2017. Architectural refactoring for the cloud: a decision-centric view on cloud migration. *Computing* 99, 2 (2017), 129––145.