Received: 22 January 2021 | Revised: 6 July 2021 | Accepted: 12 August 2021
DOI: 10.1002/smr.2378

Journal of Software: Evolution and Process
WILEY

A study of refactorings during software change tasks

Anna M. Eilertsen[1] | Gail C. Murphy[2]

[1]Institute of Informatics, University of Bergen, Bergen, Norway

[2]Department of Computer Science, University of British Columbia, Vancouver, British Columbia, Canada

Correspondence
Anna M. Eilertsen, Institute of Informatics, University of Bergen, Norway.
Email: anna.eilertsen@uib.no

Funding information
Norges Forskningsråd, Grant/Award Number: 250683

Abstract

Developers frequently undertake software change tasks that could be partially or fully automated by refactoring tools. As has been reported by others, all too often, these refactoring steps are instead performed manually by developers. These missed opportunities are referred to as occasions of *disuse* of refactoring tools. We perform an observational study in which 17 developers with professional experience attempt to solve three change tasks with steps amenable to the use of refactoring tools. We found that the strategies developers use to approach these tasks shape their workflow, which, in turn, shape the opportunities for refactoring tool use. We report on a number of findings about developer strategies, demonstrating the difficulty of aligning the kind of refactoring steps that emerge during a change task based on the strategy with the tools available. We also report on findings about refactoring tools, such as the difficulties developers face in controlling the scope of application of the tools. Our findings can help inform the designers of refactoring tools.

KEYWORDS
human factors, refactoring, refactoring tool design, software change tasks

1

1 | INTRODUCTION

Few software changes are alike: they vary in size, in complexity, and in their motivating reasons (e.g., to add functionality and to fix a bug), among other differences.[1,2] Despite this substantial variability, one similarity is that software change tasks almost always involve modification of code by a developer.

Several studies[3–6] have shown that some portion of the modifications developers undertake as part of change tasks are refactorings, namely, modifications that alter the structure of a system while preserving its behavior.[7] As a simple example, a developer may change the name of a method to better reflect its purpose. If the developer recognizes that this renaming operation is a refactoring automated by their development environment, they can invoke the automated support to update the method name *and* all references (e.g., call sites) to the new name. Common refactorings like `rename`, `extract`, and `move`[8] occur both during software changes that are purely quality-improving and—more commonly—during changes to software functionality.[5,9,10]

[1]Refactorings: For descriptions of refactorings referred to in this paper, consult the Appendix 3.

This is an open access article under the terms of the Creative Commons Attribution License, which permits use, distribution and reproduction in any medium, provided the original work is properly cited.
© 2021 The Authors. Journal of Software: Evolution and Process published by John Wiley & Sons Ltd.

The automation of refactoring operations is intended to enable developers to modify code faster and with fewer errors. This is a seductive proposition and has led to many research and development efforts aimed at enabling and supporting a broad set of refactorings in commonly used integrated development environments (IDEs). For example, the standard installations of mainstream IDEs for the popular object-oriented language Java (e.g., Eclipse[11]) and C# (e.g., Visual Studio[12]) each offer their users automated tools for over 40 refactoring operations.

Despite a long-standing prevalence of refactoring tools in development environments, it is widely recognized that this refactoring support is not used as often as would be expected, and in fact, these tools are *disused*.[13] To address this disuse, researchers have studied a variety of different approaches, including improving developers' awareness of the tools,[14–16] improving the usability[17] and correctness[18,19] of the tools, and investigating factors affecting developers' acceptance of automation.[20] These approaches generally assume that increased use of automated refactorings is largely about the tools: if developers are made aware of tools and the tools are made usable, then the tools will be used.

We take a different perspective. We believe that to be both useful and used, tools must not only automate code transformations but must fit into a developer's overall workflow. In other words, we pursue an approach that is about the humans first and the tools second. We thus choose to investigate the broader context of refactoring during software change tasks by considering how developers approach the tasks, how their approach enables or blocks opportunities to use a refactoring, and their use and success with refactorings, whether manual or automated. We perform an observational study to investigate, in a controlled environment, how 17 experienced developers approach three software change tasks that are amenable to the use of automated refactorings. With this study, we explore three research questions:

RQ1: What strategies do developers use to approach software change tasks which include refactorings?
RQ2: How often do developers use automated support for refactoring versus proceeding manually?
RQ3: How do developers experience refactoring tools?

We found that the participants made progress on the tasks utilizing one of three strategies: (1) a **Local** strategy in which the participant's workflow largely relies on frequent code changes guided by local compiler errors, (2) a **Structure** strategy in which the participant builds up an understanding of the code structure and uses that structure to guide their changes, and (3) an **Execute** strategy in which the participant guides their changes by managing and executing tests.

While refactoring tools were used by several participants from each strategy group, the tools were most commonly used by participants who employed the **Structure** strategy. This increased usage seems to be due to the tools being more aligned with this strategy. Participants using other strategies struggled to use the tools as the tools did not align with their workflows. For instance, participants using a **Local** strategy desired to follow a change step-by-step: the extensive changes supported by a refactoring tool do not align with the need for these developers to see each change as it occurs. Refactoring tools would need to operate differently for participants using the **Local** and **Execute** strategies to make use of the tools.

We also report novel findings of how developers use refactoring tools, such as gathering information from a tool's "preview view" to guide manual code changes or relying on diff-tools like git to learn about the impact that the refactoring tool had on their code. This work also provides detailed examples of previously suggested barriers to the use of refactoring tools, such as lack of trust or a gap between the operation needed in the eyes of the developer and the operations supported by tools. On the basis of our findings, we propose several changes to refactoring tool workflows.

This paper makes four contributions:

- It reports on an observational study that enables investigation of how software developers approach software change tasks amenable to refactoring. We provide a replication package to allow others to undertake a similar study or use the experimental materials for other refactoring studies (Section 8).
- It introduces a categorization of workflows developers use to approach change tasks in terms of **Local**, **Structure**, and **Execute** strategies. This categorization may be useful for empirical studies in software evolution.
- It presents a series of findings of how developers approach refactoring operations that can inform practitioners and toolmakers, such as the need to control the scope of code transformations provided by the tools.
- It demonstrates the need to study refactoring operations through the broader context of how software developers work.

We begin by placing our study in the context of earlier studies of automated refactoring (Section 2). We then describe the study (Section 3), present the results (Section 5), discuss threats (Section 6), and consider implications (Section 7). We summarize our findings in the final section of the paper (Section 8).

2 | RELATED WORK

The potential benefits of refactoring tools have been recognized for over three decades.[21] Much of the research that has been conducted about refactorings has focused on making refactoring tools correct and usable. With respect to these tools, researchers have looked at improving correctness[22] by means of different implementation techniques,[18,23–25] checking a minimal set of preconditions,[19] and removing bugs.[26] Researchers have also investigated how to increase the speed of use[27] and investigated overall usability. [15–17,28–31] The refactoring tools that exist in commonly used development environments reflect some of these research findings.

In this paper, we accept these tools as they are, focusing instead on how developers make use of them. We flip the focus by presenting developers with tasks that are amenable to the use of refactorings and observing how they proceed. With this focus, our investigation is more similar to research conducted about how developers use refactoring tools what experiences motivate use or disuse.

An early example of such a study was undertaken by Murphy-Hill et al.[6] In this study, data were gathered about developer interactions with environments that included refactoring tools, as well as from version control repositories into which developers stored code. Consideration of both tool use and version histories is common in refactoring research for two reasons: (1) it enables identification of when refactoring tools were invoked and whether that use resulted in a lasting code change, and (2) it enables mining when refactorings might have occurred based on how the code evolved as seen through analysis of the version history. Murphy-Hill et al. used this information to characterize how often different refactoring operations were invoked, completed, and integrated into the code repository. They found that 90% of refactorings were performed manually and conducted followup interviews with five participants to understand why. These participants self-reported not using tools due to lack of awareness of tools, lack of trust in tools, no opportunity to use them, and tools disrupting their workflow.

In another study, Murphy-Hill et al. investigated specific usability problems with the `extract-method` refactoring in an observational study.[17] They instructed participants to apply the `extract method` refactoring to source code while the experimenters recorded any usability problems they encountered. They observed several usability problems that could lead to tool disuse, such as the inability to select code that formed statements and incomprehensible error messages.

Vakilian et al. also investigated refactoring tool use by analyzing data from IDEs that captured developer interactions. They investigated how developers invoked, configured, and applied refactoring tools and found further evidence for disuse.[13] They conducted interviews with a subset of the participating developers (11 participants) to understand broad issues affecting the use of refactorings. On the basis of their interview data, the authors suggest factors that might impact developers' choice to use or disuse refactoring tools, such as lack of awareness of tools, lack of trust, and unpredictable tools.

Silva et al. used an interesting methodological approach to investigate why developers use refactorings.[5] They detected refactoring commits to a number of GitHub open-source projects and immediately query contributors for the motivations behind their refactoring operations. They found that refactoring activity is mainly driven by changes in requirements or *functional* changes. They also ask contributors if the refactoring was performed manually or using a tool. Over half of the respondents report doing it manually and mention several factors contributing to this choice, including lack of awareness, lack of trust, the complexity being too high or too low, and lack of IDE support.

Kim et al. were also interested in how software developers experience refactorings.[32] They conducted a survey of Microsoft employees and more in-depth interviews with a specific refactoring team and an analysis of that team's version history. Their participants self-report doing 85% of refactorings manually. In their study, the meaning of the term “refactoring” was broader, and refactoring operations like `remove parameter` were used interchangeably with large quality-improving program changes that occurred over several years. The latter category of refactorings is less amenable to direct tool support and varies substantially from the smaller operations for which automated support appears in development environments. This definition of refactoring is nonetheless common in industry and can lead to ambiguity in study results when the type of refactoring is not defined.

Liu W. and Liu H. were interested in what motivates developers to apply refactorings and focused their study on the `extract method`. Through an analysis of version histories of open-source projects and interviews of developers, they found reuse to be the main motivation for performing this specific refactoring.[33] Paixão et al. also investigated why developers apply refactorings and mined code changes from open-source communities, finding that refactorings are mainly driven by the intent to introduce or enhance features.[9]

Two other studies that summarize perspectives on refactoring from the industry are conducted by Sharma et al[14] and Leppänen et al.[34] These studies investigated both kinds of refactoring and suggest factors that deter from tool use, such as lack of exposure to tool and differential code bases[14] and lack of trust in tools.[34] Despite trust being a recurring theme in these studies, it is not well understood what developers refer to when they complain of lack of trust, nor how to address it in tools. Whereas some researchers argue that trust requires *correct* tools, others speculate that trust is tied to the tools' transparency and predictability.[13,35]

Several studies refer to the *usability* of refactoring tools without defining what usability is. We previously argued for the utility of lab studies with practitioners to investigate this facet of refactoring tools.[36] We later introduced a theory of refactoring tool usability.[37] We developed this theory by investigating the transcripts from the study reported in this paper from the perspective of usability as defined by the ISO 9241-11.[38] The ISO definition focuses on qualities of the developer's experience of the human–computer interaction that occurs

when they use a refactoring tool, such as satisfaction. In contrast, this paper focuses on understanding how the act of refactoring—with or without the use of an automated refactoring tool—fits into developer workflows. The analysis of the transcripts used to develop the theory of usability is distinct from the analysis of study data reported in this paper. The details of the study method are also unique to this paper. The theory we presented says:

> Software developers employ refactoring tools to help prepare or complete functional changes to a software system. Software developers seek these tools to be reasonable to locate, and for the tools to help them assess the efficiency of the tool, in terms of the costs and benefits of the tool, before its use. To enable effective use in multiple situations, software developers seek to guide how a tool changes the source code for a system; this ability to tailor how a tool works can improve the efficiency of the tool for the developer. Software developers also seek refactoring tools to explain their impact to source code so that the software developer can understand the effectiveness of the tool. Software developers also expect tools to communicate clearly and directly in terms that match how software developers perceive refactoring operations. These characteristics in a refactoring tool increase the satisfaction of the software developer using a refactoring tool.

The existing literature shows that developers do perform refactorings during software change tasks and often do so manually: Kim et al. found that developers at Microsoft self-report doing 86% manually,[32] Murphy-Hill et al. found 90% to be manual,[6] and both Negara et al. and Silva et al. reported more than half of the refactorings they studied to be performed manually. The existing literature then hones in on why existing tool support is disused, tacitly assuming that the automated refactoring tools are appropriate for *how* developers work. In this paper, we flip the focus, presenting developers with tasks that are amenable to the use of refactorings and observing how they proceed. Unlike previous observational studies,[16,17] we do not ask developers to perform refactorings. Instead, we ask them to perform realistic software change tasks that include code changes amenable to automated refactoring support. We then observe what they do and what they say regardless of whether they use or disuse the tools and conduct immediate follow-up interviews to discuss their experiences. With this approach, we question the tacit assumption that refactoring tools as defined are an adequate substitution for developer's manual processes and open up opportunities to consider how to help developers complete change tasks that include refactorings.

3 | STUDY

To support the investigation of how software developers use refactoring tools in software change tasks, we performed an observational study. We recruited software developers with professional experience and asked them to solve a set of software change tasks while following a think-aloud protocol. After their work on the tasks, we performed a semi-structured interview. This format enabled us to compare how multiple developers approached the same three software change tasks and compare their interview transcripts, as well as to ground the interviews in the activities observed during their work on the tasks.

The tasks were performed on a scaled-down version that we created of the Apache Commons-Lang project[39] and were modeled after commits to popular open source systems, verified to contain refactorings[3] that previous work has identified to be performed manually more often than with automated tools: `change-signature`, `inline-method`, and `move-method`. This makes the tasks realistic and relevant to understand refactoring tool use and disuse.

A full experimental package is available to support replication (Section 8).

3.1 | Participants

We recruited 19 individuals with professional development experience in a North American city using snowball recruiting seeded by company contacts known to the authors. Participants filled out an online presurvey to determine eligibility and give consent. To be a participant, an individual needed at least 1 year of professional (employed and paid) experience with Java (or a similar) programming language, be able to provide a definition of refactoring, and be able to describe a few refactoring operations.

Two of the 19 admitted individuals experienced challenges during the experiment that led us to exclude them from the participant pool and analysis: one encountered technical problems with the IDE that required stopping the experiment, and one was unable to make progress on simple tasks. The results from these two participants are not included in the analysis and results presented in this paper: in the remainder of this paper, we use "participants" to mean the 17 remaining participants and use the notation P_x to reference the experiences of a particular numbered participant. To maintain consistency with the data package, we keep the original indexing of the remaining participants. This explains the existence of P_{18} and P_{19} despite the results pertaining to only 17 participants. Table 1 presents an overview of the participants.

TABLE 1 Overview of all the admitted participants

P_{nr}	P_2	P_3	P_4	P_5	P_6	P_7	P_8	P_9	P_{10}	P_{11}	P_{12}	P_{13}	P_{14}	P_{15}	P_{16}	P_{18}	P_{19}
Exp (years)	20+	10	6	2	5	8	20+	10	10	16	11	1	20	2	10	2	12
Gender	M	M	M	F	M	M	M	M	M	M	M	M	M	F	M	M	M
Editor[a]	*IENVs*	*S*	*IE*	*I*	*IE*	*IE*	*IENVsXAs*	*IE*	*I*	*VsXc*	*IVs*	*I*	*I*	*ENAs*	*IEVs*	*IVs*	*E*

Note: Rows denote number of years of professional experience, gender, and the editors that participants reported familiarity with. Thirteen participants reported that they were familiar with the editor that was used in the study, IntelliJ.
[a] *I* = IntelliJ, *E* = Eclipse, *N* = Netbeans, *Vs* = Visual Studio, *As* = Android Studio, *X* = Xamarin, *Xc* = XCode, *S* = Sublime.

For the 17 participants, the mean (and average) number of years of work experience was 10, with the range of experience being from one to more than 20. Two participants did not have previous experience in Java but were accepted due to extensive experience (≥10 years) with other object-oriented languages. Only two of the participants presented as female. Four participants were enrolled as students. Each participant was allowed to choose between study locations hosted by the experimenter (the first author of this paper) on the university campus or to host the experimenter at their workplace. Seven participants chose the campus location, and 10 chose to provide a meeting room at their workplace. All participants received a $20 gift card as a token of appreciation.

3.2 | Experimental session

The experimental session was conducted in person with one participant at a time. We provided each participant with the same laptop, system, and tasks. The first author of this paper acted as an experimenter and was present in the room during the experiment. The experimenter administered the consent form, the tasks, answered questions, gave prompts, and took notes during the tasks. The experimenter was positioned such that she could see the screen and prompt the participant if he stopped describing what he was doing. The experimenter also made notes of events that occurred during the tasks that could be investigated further in the interview segments.

After obtaining consent, an experimental session began with an explanation of the experiment setup with an introduction to the scope of the tasks, the participant's role, and the experimenter's role. The participant was told that they would be asked to undertake three provided change tasks in succession on a codebase loaded into a development environment. The participant was told that there was no set time per task but that the experimenter would indicate if it was time to move to the next task to keep the overall session within 2 h and make time for an interview segment at the end. All experimental sessions were completed within 2 h.

The tasks were given in order, with the task description being revealed to the participant when that task began. For each task, the participant was given a task description on paper and asked to describe their plan for completing the task. Knowing the upfront plan enabled the experimenter to detect deviations from the plan and to prompt for events or experiences that triggered changes to the plan. The participant was told that they would not be bound to their plan.

The participant was asked to think aloud as they worked and were prompted if they lapsed in voicing their thoughts and actions. The experimenter answered questions asked by the participant, such as helping to orient them in the development environment. Answers related to questions about the tasks themselves were limited to contextual information from the original commits (e.g., motivation for the task) and the scope of the tasks (e.g., questions related to client code).

For each session, the participant's screen and audio were recorded, and their code changes saved. At the end of a task, the experimenter asked:

1. Which source code changes did you make in order to solve this task?
2. Do you know of any tools that could have automated any of the changes you made?
3. Are there any changes you are unsure if you got right?

The experimenter also asked for clarifications on the participant's actions and strategies. Participants might also be asked to explain specific things they had done or why they changed their plans while the task was fresh in their minds. If they expressed different strategies throughout the task, they may be asked how they would have solved it if they were to do it again. To avoid bias, the experimenter took care to not use refactoring names until after a participant did and subsequently used the name that the participant chose in conversation.

After all three tasks were over, the experimenter immediately conducted a longer interview with that participant. The interview included questions about their past experience with refactoring tools and what impacted their choices to use or not to use these tools during the tasks.

3.3 | Experimental system and tasks

The change tasks used in the study are based on actual changes to open-source systems. We sourced these change tasks from commits to open-source systems. We looked for commits that contained refactorings that have been identified by earlier research as more often manually performed and that were understandable and replicable by experienced developers in a reasonable time. We investigated the dataset from Silva et al[5] and used a state-of-the-art tool, RefactoringMiner,[3] to mine several additional repositories, including Apache Commons-Lang.

The three change tasks are based on existing commits to open-source systems: the first two from Apache Commons-Lang and one from Quasar.[40] We ensured that the descriptions of tasks did not instruct developers to perform specific refactorings, nor did the instructions include the word refactoring. Indeed, out of the three tasks, only the first task is a *pure* refactoring, that is, a code change that only alters structure and not functionality. This task involves moving methods around, whereas the other two tasks alter APIs and thus require corresponding changes to tests. Previous studies have identified such API-level changes as containing refactorings. [4]

The initial (6) pilot sessions were performed on the systems in which the commits were detected. The pilot participants solved the first two tasks in Apache Commons-Lang and the third task in Quasar. The pilot sessions revealed that it was challenging for participants to switch context between multiple software projects and to solve tasks in source code about an unfamiliar domain and with frameworks or coding style that were unfamiliar to them. We acted on this feedback by recreating a part of Apache Commons-Lang as a stand-alone system and mapping the three commits onto this system. The two commits that came Apache Commons-Lang were easy to map to our system; we adjusted the third to fit the new code context verified that the task contained the same refactorings as in the original commit by using the RefactoringMiner tool.

The target system was a scaled-down version we created of the Apache Commons-Lang project, which provides common helper methods such as string manipulation and basic numerical methods. This system is self-contained yet large enough to simulate a realistic project, comprising 78 K lines of Java, version 1.8, and 1335 JUnit tests. The system is built using Maven[41] and version-controlled through git. Participants worked on a provided Macbook Air, 13", 2017, and were asked to use IntelliJ CE IDEA 2017.3.

This system was chosen because of its relative simplicity: the target domain of Apache Commons-Lang should be familiar to most Java programmers; it is relatively simple to scale it down in size and complexity (e.g., removing external dependencies) so that an experienced developer can become familiar with the code in a short time; it is large enough that a developer will benefit from using tools to change the code; and finally, the system follows common programming patterns such as JUnit tests and Maven standard layout for files and consists mainly of utility classes with static methods. In order to solve all three tasks, participants need to change (e.g., move, alter, or delete) 45 methods distributed across 5 files. The pilot participants were not counted among the study participants.

3.3.1 | Task 1: Organize test methods

In this task, a participant is asked to reorganize 12 listed test methods from two large existing test classes into a new class, such that similar testing functionality is gathered together. This task replicates part of an Apache Commons-Lang commit[42] that reorganized their test methods.

To complete this task, a participant needed to create a new class into which the 12 listed methods were to be placed. For the code to compile, there was also a need to include the required imports and resolve a reference in the moved methods to a local constant defined in the original test class. The constant could be resolved by duplicating it in the new test class, by increasing visibility of the originally defined constant, or replacing references to it with its value.

Work on this task could benefit from the help of such automated refactorings as `move method` and `extract-class`. The mining tool shows this task as `move-method` and `extract-class`.

3.3.2 | Task 2: Removing redundant methods

In this task, a participant is asked to remove two methods, which are negated versions of two other methods in the API. The two other methods depend on the negated versions. Listing 1 shows one of the original method pairs, and Listing 2 shows part of the solution for that pair. Each of the four methods has a single JUnit test and Javadocs. This task replicates an Apache Commons-Lang commit,[43] which is discussed in this pull request.[44]

The minimal changes required to solve this task are to remove the two designated methods and their test methods and then to restore behavior in the project. Restoring behavior requires either inlining the methods before removal or reimplementing the callers.

Work on this task could benefit from the `inline-method` refactoring. RefactoringMiner shows this task as `inline-method`.

Listing 2: The resulting code after removing isAnyNotEmpty and moving its implementation to isAllEmpty.

```
60  public static boolean isAllEmpty(final CharSequence... css) {
61     if (ArrayUtils.isEmpty(css)) {
62        return true;
63     }
64     for (final CharSequence cs : css) {
65        if (isNotEmpty(cs)) {
66           return false;
67        }
68     }
69     return true;
70  }
```

Listing 1: The method isAllEmpty depends on isAnyNotEmpty. Task-2 involves removing isAnyNotBlank.

```
60  public static boolean isAllEmpty(final CharSequence... css) {
61     return !isAnyNotEmpty(css);
62  }
63
64  public static boolean isAnyNotEmpty(final CharSequence... css) {
65     if (ArrayUtils.isEmpty(css)) {
66        return false;
67     }
68     for (final CharSequence cs : css) {
69        if (isNotEmpty(cs)) {
70           return true;
71        }
72     }
73     return false;
74  }
```

3.3.3 | Task 3: Overloaded method reduction

In this task, a participant is asked to remove particular functionality from a designated class. The class consists of eight method pairs that offer functionality related to reading from and writing to objects of some data type. In each method pair, one method implements some standard behavior, and the other method takes an extra flag argument that lets callers toggle some special behavior. The first method is a wrapper to the second method, calling the second method with the extra flag argument set to a particular value. Listing 3 provides one example of such a method pair. All methods in this class have multiple tests each. Both the standard behavior and the special behavior are tested.

Listing 3: One of the eight method pairs in Task-3. The first method provides the standard behavior and the second method offers some special behavior (i.e., forcing access) if the flag parameter forceAccess is called with a `true` value.

```
112  public static void writeField(final Field field, final Object target, final Object value) throws IllegalAccessException {
113    writeField(field, target, value, false);
114  }
115
116  public static void writeField(final Field field, final Object target, final Object value, final boolean forceAccess) throws IllegalAccessException {
117    Validate.isTrue(field != null, ``The field must not be null``);
118    if (forceAccess && !field.isAccessible()) {
119       field.setAccessible(true);
120    } else {
121       MemberUtils.setAccessibleWorkaround(field);
122    }
123    field.set(target, value);
124  }
```

A participant is guided to remove the parameter such that clients can no longer detect the functionality (e.g., changing the visibility of methods is not sufficient). Each method with the flag has an overloaded version without the flag in the original code. This task replicates part of a commit to Quasar that was collected and analyzed by Silva et al.[5] We mapped the kind of code change from this commit onto a class in our experimental system and verified that it was an equivalent refactoring both manually and using RefactoringMiner.[3]

To complete this task, all eight method declarations should have the boolean parameter removed, and all calls should be updated. The method pairs should be reduced into one method, such that only the wrapper method signature exists and contains all of the logic. All logic that relies on the parameter should perform as if the parameter was false. For example, an if-branch that executes if and only if the parameter is true can be removed altogether. All tests associated with the boolean parameter are duplicated and can be removed. The many changes needed to be performed are intended to make automated support more attractive to a developer.

Work on this task can benefit from the `safe-delete`, `inline-method`, or the `change-signature` refactoring. RefactoringMiner shows this task as `inline-method`.

4 | DATA AND ANALYSIS

The study data comprised approximately 32 h of study sessions. We transcribed the audio portion of the video captured from the 17 experimental sessions and created one transcript for each session from the recorded screen capture and audio capture information. In these transcripts, we marked the sections related to interview questions and answers. In total, the 17 transcripts comprise 134,947 words and can be accessed in the replication package along with the screen recordings, participants' solution source code, and a report from the mining tool on refactorings that can be detected in each of their solutions. This data are analyzed in several steps.

The experimenter (the first author of this paper) performed the initial labeling of the transcripts for easier retrieval of points of interest by marking sections that pertain to events in the videos or comments made by participants. The labels are shown in Table 2 and identify actions or events that were significant to the participants' individual workflows, such as their stated *plans* for how to solve the tasks, which refactoring tools they *named* or *invoked*, other tools they invoked, and actions they took to *validate* that their changes were correct before continuing their work. While labels pertaining to plans or sentiments could be identified solely by the transcript and labels pertaining to invocations solely by the video, the remaining labels needed to be determined by reviewing both audio and video. The labeling was discussed with, and reviewed by, the second author of this paper.

We then created summaries of the plans and actions that each participant used to approach each task. In order to develop a schema that captured a participant's approach to each task, we included in the summary schema the following information: quotes that indicate their plans or *intent* (if any such were made); a natural-language summary of their *approach*, written by the experimenter; which refactoring tools they *mentioned*, *tried*, and *used* with and without prompts; other tools that they used to solve the task; how they guided their changes (e.g., emphasis on navigating to and fixing compiler errors, avoiding errors, and top-down in the file); and how they verified their changes. The set of possible refactoring tools is taken from the IDE that was used.[45] If a participant mentions or invokes a refactoring tool, then we consider them *aware* of the tool; if they invoke the tool, then we consider that they *tried* it; and in order to *use* a refactoring tool, the participant needs to both apply it and continue working with the resulting code.

TABLE 2 Labels used for initial labeling of transcripts

Code	Description
Plan (keywords)	The participant expresses actions they intend to take to solve the task. Keywords include short, descriptive statements such as break-fix or systematic.
Change Validation (Method)	The participant takes an action to validate their change. Method is a short, descriptive statement of the method they employed such as static (e.g., compiling, syntax errors, and warnings), dynamic (e.g., running tests), and compare (e.g., using git diff of comparing to commented out code).
Named (Refactoring)	The participant names a refactoring tool. Refactoring tools include the common refactoring tools: Rename, Move, Inline, et.c.
Invoked (Tool)	The participant invokes a refactoring tool. Tools include the common refactoring tools: Rename, Move, Inline, etc. as well as Find Usages, etc.
Restart	The participant restarts the task.
Mistake	The participant makes a coding mistake such as changing the wrong code location.
Error (Not) Understand (Refactoring/Other)	The participant encounters an error that they do (not) understand, either arising from a refactoring too, the compiler, or tests. Mark errors that are significant to their work, not all encountered errors. Mark source of error (e.g., Move and compiler) where nonobvious from transcript.
Trust	The participant expresses a sentiment related to tool use or disuse and trust or lack of trust.
Predictable	The participant expresses a sentiment related to tool use or disuse and predictability or lack of predictability.
Refactoring Insight	The participant makes an insightful statement regarding tool use or disuse that may be useful to retrieve later.

The experimenter (the first author of this paper) created the summaries by applying the schema to all participants' workflows on each task. The schema was developed by the two authors of this paper by reviewing several video recordings and discussing participant workflows. The experimenter reviewed the video and audio recording of each participant to create a summary for that participant. Figure 1 shows the schema completed for a participant working on the second task. In this summary, the participant's *intent* is represented by a quote that summarizes their intended plan and then the experimenter filled out the *Approach Summary* describing what actual actions the participants took, problems they encountered (tool error), and how they organized their changes ("Isolate changes and keeps tests running by introducing a shared dependency as an intermediate step."). Then records about refactoring tools are listed, such as tools the participant *mentions*, *tries* to use, or *uses*, as well as other tools they relied on, how they guided their changes, and how they verified their changes. To verify that the summaries appropriately capture a participant's workflow, the second author of this paper reviewed several summaries, comparing them with the video recordings. The two authors also discussed any difficult to interpret actions by developers. This process results in 51 summaries.

Once the set of 51 summaries was created, we compare them and look for patterns in the following information: the *intent* that participants express and the actions they used to *change* the code, *guide* their changes, *verify* their changes, and *recover* from unwanted errors. On the basis of the patterns between summaries, we identify different *strategies* that participants employed in each task.

5 | RESULTS

Overall, participants were able to make good progress on the tasks: all participants made some progress on all three tasks. Furthermore, all participants completed Task 1, 16 (94%) completed Task 2, and 12 (70%) completed Task 3. Twelve (70%) of participants completed all tasks. The progress made by participants on the tasks provided substantial opportunities to observe the workflow and tool use during each task. We describe our results in terms of the research questions posed in Section 1 and use the notation P_x to reference to the experiences of a particular numbered participant.

We consider the three research questions outlined in Section 1 in turn.

5.1 | RQ1: What strategies do developers use to approach software change tasks which include refactorings?

The participants took a variety of approaches to the three software change tasks. Applying the analysis described in Section 4, we identified three distinctly different strategies for interacting with the code:

1. **Local**, in which participants guided their changes by *local* and immediate information such as feedback from the compiler.
2. **Structure**, in which participants guided their changes by code *structure* such as callers, conditional branches, and references.

Intent	"My long-term goal is that I want to inline this method. So by keeping this kind of stub implementation of it - stub isn't the right word - I'm preserving it as long as possible so any tests calling it is still working while I do my refactoring. What I want to do is move the implementation into the method I am going to keep and ensure that the tests are all passing and then remove the method I want to remove."
Approach Summary	Forms plan upfront that includes Inline Method. Changes plan when encountering tool error. Isolate changes and keeps tests running by introducing a shared dependency as an intermediate step. Manual inline, keep old code in method to validate (compare). Validates change by running tests and comparing changes in git. Guides steps by Find Usages.
Refactoring Tools	
Mentioned:	Inline Method
Mentioned prompted:	
Tried:	Inline Method
Tried prompted:	
Used to solve task:	None
Other Tools:	Find Usages
Guide changes:	Find Usages, Avoid compiler errors
Verify:	Dynamic after each change, git diff

FIGURE 1 Task analysis schema for P_9 on Task 2

3. **Execute**, in which participants guided their changes by *executing* the test suite.

Figure 2 shows how the strategies used distribute across each participant's approach to a task. Each of the strategies was successfully employed by one or more participants to solve each task, indicating that a task did not dictate which strategy to use; instead, the choice of strategy was dependent on the combination of the individual and the task, with some participants changing strategies between tasks. Figure 3 further shows that there was no one dominant strategy used for a task.

Despite the variety of strategies, participants using different strategies still often produced the same source code solution in the end. For example, an **Execute** participant may first isolate the functional changes that needs to be performed in order to achieve the task goal and then update the code to match this new behavior. Meanwhile, a participant with a **Local** strategy may first alter the code that seems to implement the exact behavior that is required to change, then—almost accidentally—propagate the change to tests by iteratively fixing compiler errors. A partici-

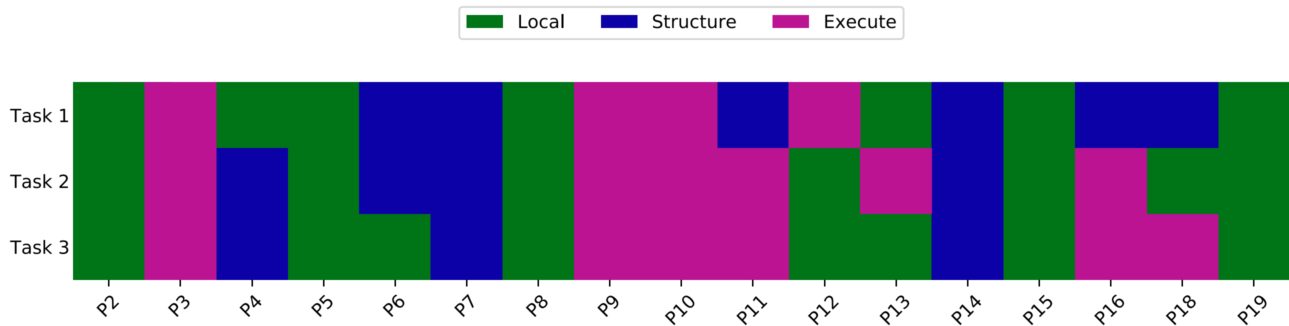

FIGURE 2 The strategies were distributed both across participants (*x*-axis) and tasks (*y*-axis). In total, 22 workflows were identified as **Local**, 13 were identified as **Structure**, and 16 were identified as **Execute**. As can be seen in P_4, P_{16}, and P_{18}, individual participants sometimes changed strategies between tasks

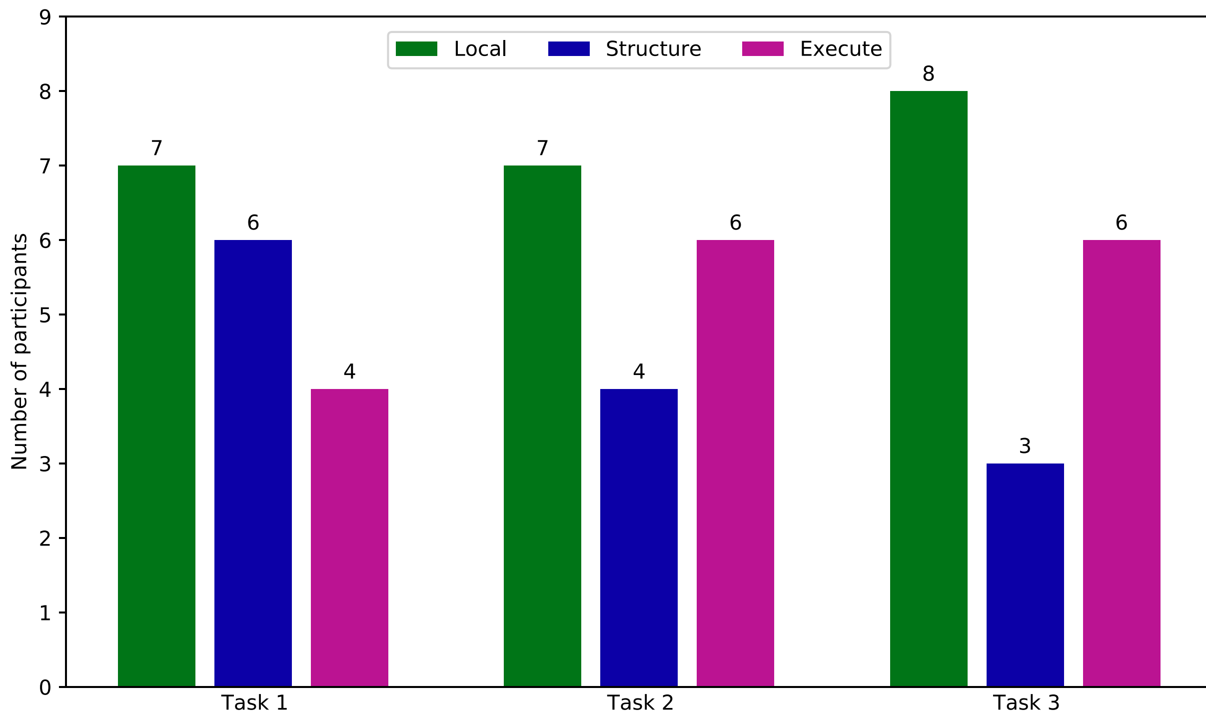

FIGURE 3 We group the workflows first by task, then by strategy, so that the 17 workflows from Task 1 is represented by the three leftmost bars, the 17 workflows from Task 2 by the three middle bars, and the 17 from Task 3 in the three rightmost bars. Within each group, the three different-colored bars indicate how the strategies were distributed across participant's workflows for that task. For example, in Task 1, there were seven participants who employed a workflow that matched the **Local** strategy, six participants with a workflow matching the **Structure** strategy, and four matching the **Execute** strategy

pant using a **Structure** approach will likely make the same code changes but may make them by studying program structure rather than isolating either local changes or changes oriented to the program execution.

5.1.1 | Local

Participants who used a **Local** strategy had workflows that were characterized by interacting predominately with information that was immediately available to them (i.e., *local* information) and changing elements that were present in their editor. Their workflows included actions that intentionally provoke feedback from the IDE such as deleting or commenting out code elements that have references elsewhere in the code, thereby introducing compiler errors or "red lines." When stating their intent, participants made comments such as P_4 on Task 1: "copy-paste them all from one place to another and see what breaks" and P_2 on Task 3: "So what I'm going to do now is rely on my friendly compiler to do this. So now things are going to break." Their workflows were oriented around introducing, navigating between, and fixing this feedback. As a result, the source code frequently underwent long periods in noncompiling states, and the impact that source code changes had on their tests in Tasks 2 and 3 was often not known until at the very end.

For example, in Task 2, participants who used a **Local** strategy in this task (P_2, P_5, P_8, P_{12}, P_{15}, P_{18}, P_{19}) began by deleting the method that contained the implementation, which leads to compiler errors in the caller. In order to "fix" the compiler error, one needs to either obtain the previously deleted code or reimplement the method from scratch. Only three of the seven participants obtained the previous implementation (through git, undo, or comments), whereas the remaining four reimplemented the method from scratch. All of these four spent significant time on resolving bugs or errors that appeared in their reimplementation. In comparison, only one other participant with a non-**Local** strategy (P_{16}) had a similar problem.

Participants who relied on the **Local** strategy were able to guide their navigation and changes by utilizing compiler errors and warnings. Yet, when they encountered errors that they did not understand, they had little choice but to "backtrack" and redo the task, hoping that their guidance would work better the next time. For instance, P_2 had to restart Task 3 after encountering a compiler error that they did not understand. This error originated from a binding change of an overloaded method that they had not noticed. They said "Just confused by this error. Wrong first argument type. That's weird. (…) So I don't understand the error." Since the participant did not understand the error or how to solve it, their solution was to restart and redo the previous steps in another order. They did not run into the same error again. On the same task, P_6 decided where to work based on how many compiler errors arose and described that they would monitor the number of red lines that a change induced and if above a certain threshold, they would undo that change and look for any other place to work.

5.1.2 | Structure

Participants who used a **Structure** strategy were orienting their workflows around source code structure, such as considering references or callers to an element. When starting a task, they typically explored code from one or more starting points using structural queries before altering the program. The understanding a participant gained about the code structure through their upfront explorations was used to formulate plans for their changes according to the impact it would have on other places in the code.

In Task 2, four participants (P_4, P_6, P_7, and P_{14}) used this strategy. All of them formed plans to inline the method that was to be removed to its caller and executed this plan either manually or using tools. For example, P_7 stated in his upfront plan that "Any invocations would need to be dealt with. So what I'd like to do, is look in StringUtils, look where these are used (..) and then depending on the implementation of the anynot I may just be able to inline them." When they changed code, they intentionally propagate changes across code structures by means of navigational tools or refactoring tools. Where other participants might organize their changes so that they can act on local information (**Local**) or around the execution of tests (**Execute**), these participants made efforts at organizing changes along structural edges even when it required more navigation or "leaving behind" problems in the code.

Both users of the **Local** and **Structure** strategy delayed test changes to the end of Tasks 2 and 3. This may not be intentional—participants did not express such an intent—but may be simply due to the compiler errors they needed to navigate during the tasks. This is in stark contrast to participants who used the **Execute** strategy who relied heavily on the tests' behavior throughout the entire task.

5.1.3 | Execute

Participants using an **Execute** strategy used a workflow that was oriented towards executing the tests and, thus, keeping the code compiling so that the tests could be run throughout the task. At the beginning of tasks, they explored the codebase using structural navigational tools, similarly to someone using **Structure** strategy, but for the purpose of locating the relevant tests and exploring the call graph.

These participants searched for a start location that allowed them to isolate functional changes from syntactic changes. For example, for Task 3, many of these participants began by locating the program element responsible for the behavior of interest and, rather than removing it, made a controlled change to the functionality while preserving program structure. The controlled change allowed them to run the test suite and locate the tests impacted by the purely functional change. In this way, the participants could safely redefine the test suite to the target state of the software change before making the structural changes to the code. For example, P_{11} described a strategy for Task 3 as follows:

> One strategy would be to change the internal definition of the methods first so that it always executes as if it was false. That will reveal the test cases that I need to look at more closely.

In contrast, participations using **Local** or **Structure** strategies typically relegated interacting with the test code until the very end. In order to keep the tests running, these participants traversed the call graph to locate "leaf nodes," nodes that do not call other methods. This helps them limit the impact of their code changes and test each change before moving on the next. For example, P_{18} used this strategy on Task 3 and noted,

> If I change it in a method where it is being passed down, it is going to lead to a lot of cascading effects, that are going to have to change other methods […]. Whereas if I do it just here, I am going to only focus on this method and worry about everything else, like up top, later.

5.1.4 | Summary and relationship to earlier work

We described three strategies that participants used to approach tasks: **Local**, **Structure**, and **Execute**. In considering how developers approach a change task, these findings are similar to those who have studied general, non-refactoring specific change tasks. Several of these earlier works have reported the use of structural investigations as a strategy for pursuing a task (e.g., Ko et al.[46]). Fewer of these studies have described a focus on the execution of the system as a strategy; the closest we are aware of is Maleej et al.'s report on the use of a debugging strategy by industrial developers.[47] We are not aware of descriptions of the use of a strategy similar to **Local** for solving tasks; however, developers have reported avoiding refactoring tools in favor of compiler errors.[6,13,16] This study is the first to our knowledge that observes these different strategies on the same tasks.

5.2 | RQ2: How often do developers use automated support for refactoring versus proceeding manually?

We investigated how often participants used refactoring tools to solve tasks. We analyzed 51 task workflows, one for each participant working on each task. Recall that we consider a participant to *use* a refactoring tool if they apply it and continue working with its output.

Of the 51 workflows, 17 workflows (33%) contained one or more recorded *uses* of refactoring tools. In the remaining 34 workflows (67%), participants solved the tasks without using refactoring tools. As seen in Table 3, this was true both for more and less experienced participants. Even in an environment where experienced developers are primed to use refactoring tools, and the tasks are amenable to use, participants did not use refactoring tools extensively.

Each task in our study included steps amenable to consideration as refactorings (Section 3.3), which means that when a participant does not *use* a refactoring tool on a task, it is an occurrence of tool *disuse*. Table 3 shows how the refactoring tool use and disuse are distributed across

TABLE 3 Overview of all the refactoring tools that participants used to solve tasks and the experience of each participant

	P_2	P_3	P_4	P_5	P_6	P_7	P_8	P_9	P_{10}	P_{11}	P_{12}	P_{13}	P_{14}	P_{15}	P_{16}	P_{18}	P_{19}	% use
Exp (yrs)	20+	10	6	2	5	8	20+	10	10	16	11	1	20	2	10	2	12	-
Task 1	-	-	-	-	ExCon	InCon	-	-	-	-	RC	-	MM	-	-	ESc	MM	35%
Task 2	-	-	-	-	-	ELVar	-	-	IM	-	-	-	ELVar	-	-	-	-	18%
									ELVar									
Task 3	-	-	ChS	SD	-	IM	ChS	ChS	-	-	-	-	IM	-	ChS	-	ChS	47%
						ChS		IM					ChS					

Note: A participant used a tool if they continued working with the output of the tool. ExCon = Extract Constant; InCon = Inline Constant; RC = Rename Class; MM = Move Method; ESc = Extract Superclass; ELVar = Extract Local Variable; IM = Inline Method; ChS = Change Signature.

participants and tasks. We can see that all tasks had three or more participants that used refactoring tools, showing that each task was indeed amenable to the use of refactoring tools. Furthermore, we observe that different participants used refactoring tools on different tasks, indicating that participants were not just invoking tools at every opportunity.

We also investigated whether tool use occurred more often in workflows that followed certain strategies. Because participants sometimes change strategies, we consider, for each participant's workflow on each task, which strategy they employed and whether they used refactoring tools or not. Figure 4 shows how tool use and disuse occurred in workflows from each strategy. We see that the workflows in which a participant employed a **Structure** strategy more often include refactoring tool use.

We also considered whether the participants who did not use tools had created solutions that were not amenable to automated refactoring tools, which would mean that these participants did not encounter opportunities to use the tools. We ran the RefactoringMiner[3] tool on all participant's code solutions and identified 385 refactorings in the code they produced. These were distributed across all participants' code solutions for all tasks. This means that every participant solved every task by changing the code in ways that were amenable to a refactoring tool being used. By comparing these results to actual use, we observe that even some participants who did use tools missed several opportunities for use by solving parts of the task manually. We describe this further when answering RQ3.

We also collected all of the refactoring invocations that participants performed throughout all three tasks and found that in total, they invoked refactoring tools 100 times. Figure 5 shows how the invocations are distributed across participants. We see that even participants who did not use refactoring tools tried to invoke them, which invites the question of why these participants did not end up using tools.

5.2.1 | Summary and relationship to earlier work

We observed whether participants used automated support for refactoring or proceeded manually. In the 51 different cases that the three tasks posed across all 17 participants, we found that in 17 (33%) cases, the participants *used* a refactoring tool, whereas in the remaining 34 cases (67%), participants solved the task by changing code manually. These numbers are aligned with previous findings in which developers self-report to use tools less than half of the time.[5,13,32,48] These studies investigate various granularity of changes: our cases are modeled off of commits and are therefore most comparable with studies like Silva et al.[5] However, by analyzing the refactoring opportunities that were identified in participants' source code, as was done by Murphy-Hill et al,[6] we find that even participants who *use* tools on a task missed a number of opportunities for tool use by solving some of it manually.

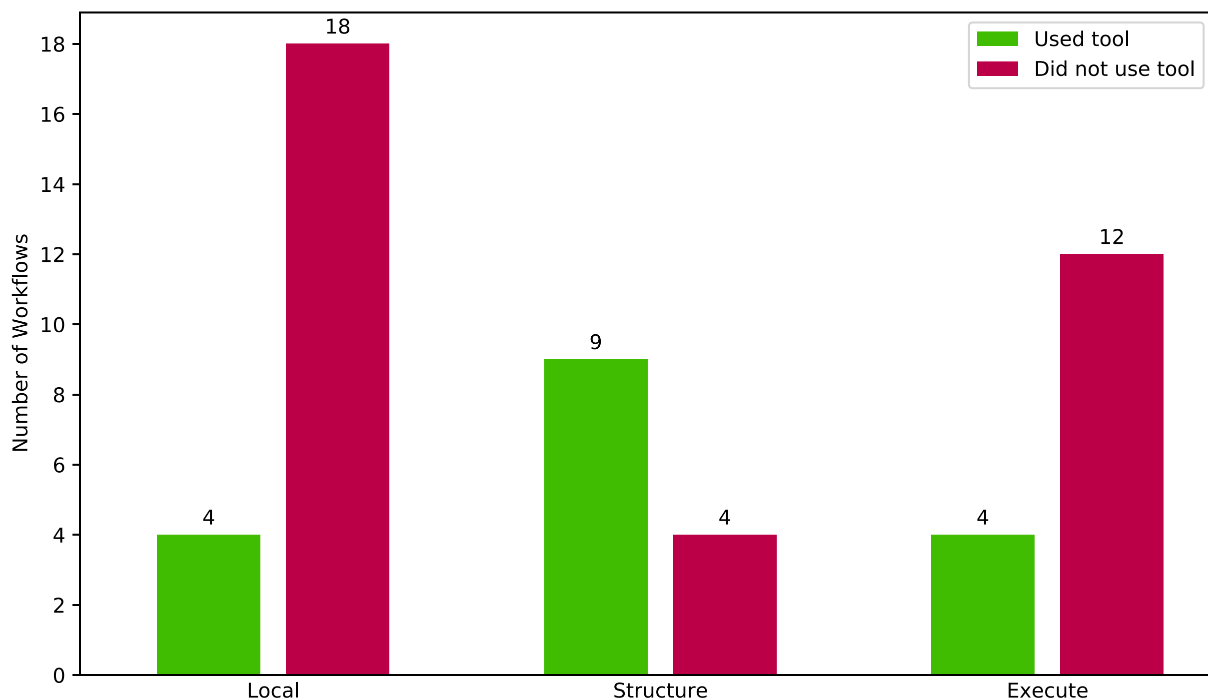

FIGURE 4 The number of times a participant employed each strategy in their workflow and used or did not use a tool in the same workflow. Each participant solved three task, for a total of 51 workflows

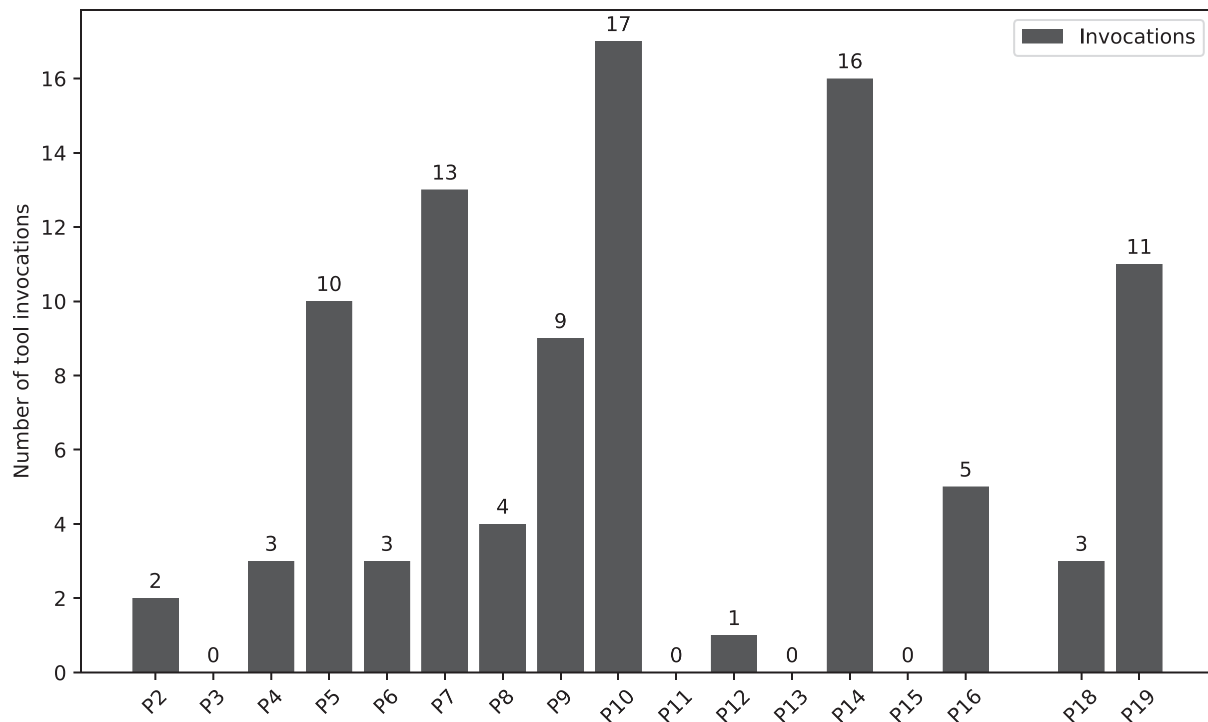

FIGURE 5 The number of refactoring tool invocations that each participants performed across all tasks

5.3 | RQ3: How do developers experience refactoring tools?

We investigated how participants experienced the refactoring tools that they interacted with during the study. Previously, we reported cases in which a participant *used* tools; here, we also consider cases in which the tool was not used, in particular where participants had experiences that led them to avoid the tool or not retain its effect on the code.

The participants who *used* a refactoring tool had to go through the following process:

1 they must be be *aware* of the tool (e.g., by recalling a tool that can help or recognizing that a tool might be available even if they do not know the name of the tool),
2. they must *try* to use the tool (e.g., by locating and invoking it),
3. they must successfully apply the tool to the code (e.g., avoiding errors and not clicking "cancel"), and
4. they must continue working with the tool's output (i.e., not reverting the application).

Many of the participants who did not use tools also went through some of this process but had experiences in various stages of this process that ultimately led them to not use tools.

For each task, we consider a participant *aware* of a refactoring tool if they mention refactoring tools that can help them during the task or subsequent interview segment or if they try to invoke one. This means that a participant may be aware of a tool that may help them in one task while not in another task. We consider that a participant *tried* a tool if they invoke it, regardless of whether the tool produces errors, they cancel the invocation, revert what the tool did, or continue working with the resulting code. Only in the latter case do we consider the tool *used*.

Figure 6 shows, for each participant and each task, whether the participant indicated *awareness* of refactoring tools, whether they *tried* invoking a refactoring tool, and whether they *used* a refactoring tool. The figure shows that most participants showed awareness of refactoring tools during the study, but far fewer ended up using one. In fact, in all tasks, there occurred a drop-off from awareness of refactoring tools to the actual use of the tools. For Task 1, 12 (71%) of participants showed awareness of a refactoring tool, but only six (35%) of participants used such a tool. For Task 2, six (35%) of participants showed awareness, and three (17%) exhibited use, and for Task 3, 14 (82%) showed awareness, and eight (47%) made use of a refactoring tool. Only three participants (17%) made use of refactoring tools in two tasks (P_7, P_{14}, and P_{19}) despite all but one participant indicating awareness of refactoring tools.

In total, out of the 51 opportunities for tool use that the 17 participants had across all three tasks, participants expressed awareness of tools in 32 (63%) cases, and out of these 32 cases, participants tried a refactoring tool in 23 cases (72%). Out of the 23 cases in which participants tried

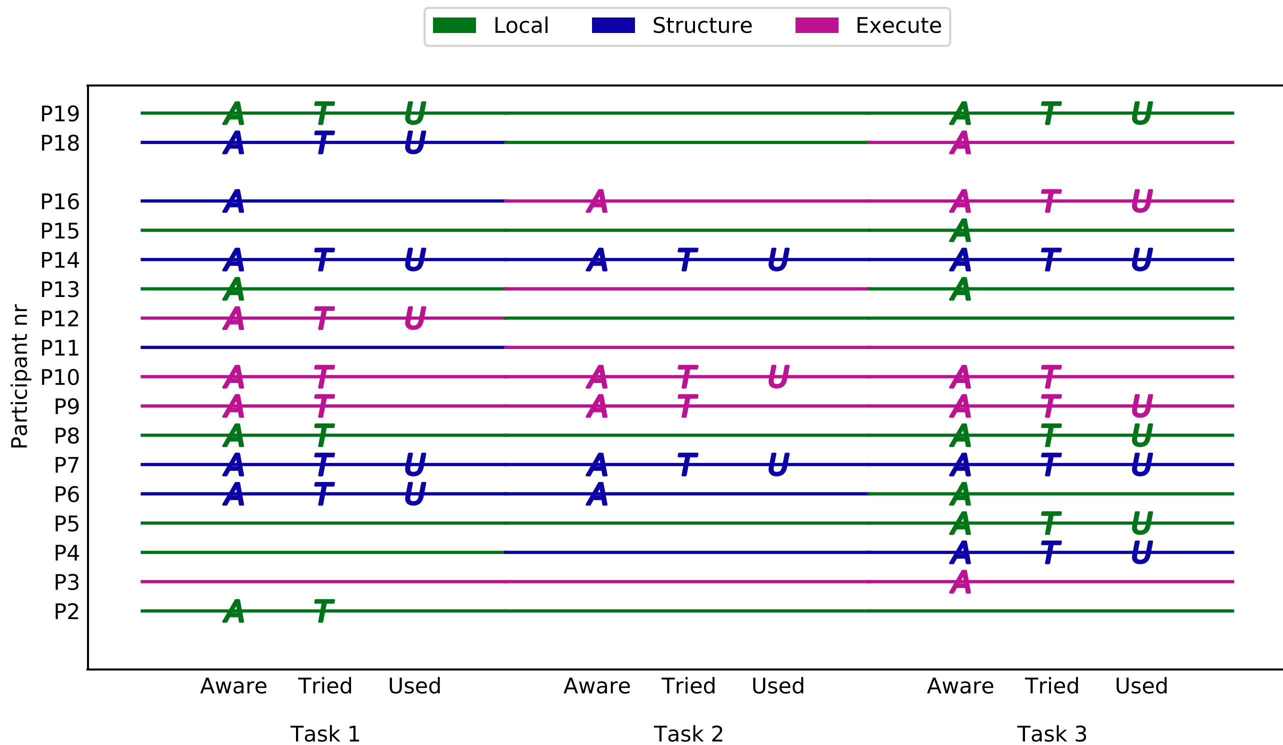

FIGURE 6 Each task presented each participant with the opportunity to use refactoring tools, for a total of 51 opportunities. In each of these 51 cases, we investigate whether the participant was *aware* of refactoring tools, *tried* to use tools, and *used* a refactoring tool once or more

a refactoring tool, they ended up using a tool in 17 (74%) cases. We see that for every step, there is a drop-off of around 30 % resulting in tools being used in only 33% of cases. To understand the drop-off from awareness to attempted use and from attempted use to use, we inspect the transcripts, including the answers to interview questions, to learn why participants whose to not continue the process of tool use. Similarly, we investigate, for participants who used tools, how they experienced them. In each of these steps, we observed a number of factors that impacted how participants experienced the tools.

5.3.1 | Awareness

Participants were mostly aware of refactoring tools. In Task 1, 12 participants (71%) showed awareness of tools, all of whom indicated that they could use `move` or `extract`. In Task 2, six participants (35%) showed *awareness* of tools, five of whom indicated that they could use `inline-method`, and one participant (P_6) mentioned `safe delete`. In Task 3, 14 participants (82%) indicated awareness and referred to `change-signature`, `inline`, or `safe-delete`. All three tasks presented the 17 developers with 51 cases in which they had an opportunity to use a refactoring tool, and participants were *aware* of tools in 32 (63%) cases. From these 32 cases, participants moved on to *try* a tool in 23 cases (72%), while in the remaining nine cases, they did not. We observed multiple reasons why the remaining participants decided not to even try the tool to help with a task. The main reason given by participants was that they had proceeded too far in the change before realizing a tool could help them. Another recurring reason was that the tool might negatively impact later steps in their workflow.

In Task 1, P_{13} expressed that they realized after the fact that they could have used a tool, whereas P_{16} did look for a way to invoke *move* from the "structure" view, which shows a truncated view of the class by listing class member declarations such as methods and fields. They failed to do so and said, "I spent a couple of minutes and didn't get it working. I decided it is faster to do manually."

In Task 2, P_{16} made the manual change before realizing that they could use `inline`, and P_6 did not invoke `safe-delete` because they did not "trust" the tool. When asked to explain, this participant expressed that they did not trust that the tool would present them with information about the code that would teach them as much as they would learn by manually changing it: "I prefer to see these things happening step by step because I know what's there, I can double-check any of the files that are to be changed and see if there are any other changes that need to be done in them."

In Task 3, five participants (P_3, P_6, P_{13}, P_{15}, P_{18}) did not try refactoring tools despite being aware of them. P_3 mentioned `inline` but said they would look for an automated approach only if they wasted too much time doing it by hand and if they trusted their tests. P_{13}, P_{15}, and P_{18}

named `change-signature` during the interview but said that they were not sure how much it would have helped them. P_{13} and P_{18} also did not realize upfront that they could use this tool, whereas P_{15} feared that it either would not help them or impact code that they did not yet want to change:

> Change signature might have helped you but not really. It would tell me that the method with the signature which is without the boolean already exist. So not really. I also don't want to remove all the callers of the method because I want to look at them and see whether or not it makes sense for them to be removed or whether they should stay (P_{15}).

A similar sentiment was expressed by P_6, who mentioned `safe-delete` but said they did not trust the tool and explained: "I would have the problem of doing a refactoring and not knowing if the behavior of the test should be the same or the behavior of the code." They added, "I don't want to rely on a tool that will not help me become familiar with the source code," explaining that they gained an understanding of the code by making changes manually. If they were to make changes in the future, they would be faster due to this experience, while using a tool would deprive them of gathering similar knowledge.

In summary, we found that participants who were aware of tools without trying them experienced the tool as unable to present them with satisfactory information about the code and unable to distinguish test code from functional code. Both of these problems were noted by participants as making later steps in their workflow more difficult, both during the same task and if they were to work with the same codebase again. These experiences have not, to our knowledge, been previously reported. We also observed experiences that support previously reported factors that motivate disuse, like realizing too late that a refactoring tool can be used[6] and lack of trust.[5,6,13,34]

5.3.2 | Tries

In the 23 cases where participants *tried* refactoring tools, they made one or more invocations of the tools that are listed in Table 4. In 17 of the cases (74%), this resulted in *use*, whereas in the remaining six, participants did not end up using any refactoring tool on that task. We had expected that participants who experienced so-called *barriers to use*[17]—warnings, problems, or errors—might avoid using tools; however, although participants did encounter such barriers in these six cases, so did the participants who went on to use tools. In fact, in every task, most of the sparticipants who *tried* a refactoring tool encountered some variation of tool warnings, problems, errors, or the introduction compiler errors, and while these experiences led participants to abandon the tool in Tasks 1 and 2, participants repeatedly relied on tools in Task 3, despite encountering barriers to use. In some cases, these were even the same participants.

In Task 1, P_2 and P_{18} had experiences when invoking the tools that led them to avoid using them. Both participants tried the `move` refactoring but had problems invoking the right version of `move` due to incorrectly assuming that the tool would understand their code selection as arguments to the refactoring. Unlike previously reported problems with selecting statements correctly,[17] these participants did make a correct selection of the methods they wanted to move in the editor. Then they right-clicked somewhere on the selected text and invoked `move` in the refactoring menu, thereby indicating that they wanted to `move` all of the selected methods. However, the tool failed to recognize the selection altogether and instead interpreted the invoking to apply to the surrounding class, thereby launching `move-class` instead of `move-method`. Both participants repeated this interaction several times before giving up and indicating that they might as well move the methods manually, referring to the simplicity of the task.

In Task 2, all participants who tried a refactoring tool (P_7, P_9, P_{10}, P_{14}) initially invoked `inline-method`. These participants experienced a problem that has previously been reported for other tools: misinterpreting error messages.[17] When invoking `inline` on the method to be inlined, the tool produced an error due to this method's code structure. All four participants misinterpreted the error as relating to the structure of the caller. As a result, all four participants performed `extract-local-variable` in the caller in an attempt to bypass the error, with no success. By

TABLE 4 Summary of refactoring tools invoked during the study

Refactoring	Code element type	Count
Change signature	Method	36
Inline	Constant, method	25
Move	Instance method, static method, class	19
Safe delete	Method, parameter	12
Extract	Constant, local variable, superclass	7
Rename	Class	1

Note: The refactoring, on the left, was invoked on the code elements seen in the middle. For example, both `inline-constant` and `inline-method` were invoked.

the time they understood the problem, three of them had decided it was more efficient to manually inline it than to fix the error. Only P_{10} performed another `extract-local-variable` correctly in order to benefit from the tool.

In Task 3, all participants experienced problems with colliding method signatures, the parameter they wanted to remove being in use, or the same problem with `inline-method` as described for Task 2. Five participants (P_4, P_5, P_8, P_{10}, P_{19}) initially canceled the refactoring tool invocation in response to problems and inspected or edited the offending code in the editor before invoking the refactoring again, whereas four participants (P_7, P_9, P_{14}, P_{16}) accepted the problems upfront and applied the tool. In this task, most participants accepted that the refactoring tool took them "part of the way" so they could complete the change themselves, similar to workflows suggested by Vakilian et al.[13] All of the participants who tried a tool on this task eventually applied it, despite initially encountering problems.

The only participant who *tried* a tool without *using* it in Task 3 was P_{10}. They invoked `change-signature` and `safe-delete` several times but decided to approach the task manually to stay in control of the change. This participant first spent a few tries on getting configuration options right for `change-signature` before realizing that the tool also updated test callers. Upon discovering this lack of separation between tests and production code, they said, "I don't want to like change my production code and change my test and then run my tests and see if everything is fine." They restarted the task and tried `safe-delete` but eventually decided to solve it manually. When prompted to explain, they said, "My fear is that safe delete would delete anything that calls it."

In summary, we observed that participants who tried tools experienced novel problems such as incorrectly assuming that the tool would understand their selection as the refactoring argument and problems controlling the impact of the tool. The latter experience lends credibility to participants who avoided trying tools in order to not have them impact test code. We also observed previously reported experiences, like misunderstanding error messages[17] and encountering violated refactoring preconditions.

Furthermore, we observed that when participants encountered problems, they decided to use or not use tools based on a *cost–benefit analysis*. By evaluating whether the perceived *benefit* that the tool provided outweighed the perceived *cost* of overcoming these problems, they decided to pursue solutions to problems that they encountered or to switch to a manual approach. Participants seemed to evaluate these costs and benefits differently: for example, P_{10} was the only participant who only *tried* a tool in Task 3 and decided that the cost outweighed the benefits. This analysis is also dependent on the difficulty associated with performing the refactoring manually. Participants in the first two tasks indicated that the benefits of using tools in these tasks were low due to the perceived simplicity of the code change and the low number of repeating changes. In these tasks, when the cost of using the tool increased, such as when encountering problems or failing to invoke the right tool, the threshold for favoring a manual approach was low. In contrast, the third task was more complicated and required a higher number of repeating changes, which motivated a higher number of participants to "pay the cost" by figuring out how to overcome these barriers.

This is the first study that observes developers evaluate the cost and benefit of using refactoring tools on the fly. Our earlier paper identifies this as one of four usability themes.[37] However, our findings echoes a theoretical perspective from Fleming et al., who employed information foraging theory in a theoretical analysis of refactoring tools used for smell removal.[49] Our findings provide empirical evidence for Fleming et al.'s prediction that, in cases where tools may have difficulty accurately approximating the human judgment required to determine whether individual code changes are appropriate, the cost of using automation increases. Similarly, in cases where the tool presents the user with information that they otherwise would have to forage for, the tool decreases the overall cost involved in performing the refactoring.

5.3.3 | Use

Participants *used* refactoring tools in 17 workflows (33%) with varying results. In some cases, participants applied the tool with ease, such as for `rename`, `extract-local-variable`, `extract-constant`, and `inline-constant`. However, in the case of more complex[13] tools, such as `change-signature` and `move-method`, some participants made the tools work for them, whereas others reverted to a manual approach. Similarly to participants who *tried*tools, these participants expressed that they evaluated whether the benefit of automation outweighed the cost of using it. Some factors that impacted this cost–benefit analysis were the tool's impact on test code, difficulties understanding the tool's impact on code, and code layout. The participants who successfully used tools were careful with the order in which they applied refactorings or used the information presented by the tools to organize their manual changes.

In Task 3, participants experienced problems with the refactoring tool's impact test code. In this task, participants mainly invoked `change-signature`, `inline-method`, or `safe-delete`, tools that propagate changes to all callers. Many participants organized their code changes in the source code file separately from the test file. However, refactoring tools do not distinguish code from tests and treat all as one program, leading to callers in the test code being changed as well despite participants invoking the tool in the source code file. While this is a natural and correct behavior from the tool, it was unwelcome by many participants. We already saw that several participants who avoided trying and using tools in Task 3 mentioned this problem. Several of the participants who used tools (e.g., P_4, P_9) realized its impact on test code after applying a tool and refrained from using the same tool again. Other participants (P_5, P_7, P_{19}) only realized this impact once they began working on the test code at which point they were unwilling to revert all their work and instead utilized git diff to learn about how each location had changed. One of the participants who used `change-signature` throughout all of Task 3, P_{19}, gave this response when asked whether the tool did what they wanted:

> Not completely. I think if I'd played a little more I could have excluded part of the refactoring. I'd like to say I want to go ahead with the refactoring in this area and not in this area, because if I could exclude the tests I could better see the impact of what tests are actually broken. But you want to have the proper changes done inside the code (P_{19}).

The problem with test code highlighted the participants' dependence on diff tools to understand the tool's impact: out of the nine participants who tried a refactoring tool in Task 3, seven participants (P_4, P_5, P_7, P_9, P_{10}, P_{16}, P_{19}) resorted to git to disambiguate callers both in the source code file and the test file after applying the refactorings. These participants inspected review views and problems views before applying the tool, yet found that afterwards, they needed to know what the code used to do or how it had changed. Several participants expressed dislike over this workflow. P_9 and P_7 made the following comments:

> Some of these tests should be failing because they should be testing behavior that is no longer supported. So I should go through and look at the tests. Maybe I should have done this before deleting the parameter because now it's going to be hard to look at the tests and know what they were doing before. Oops (P_9).

> By invoking the refactoring tool it changed code I wasn't looking at. So a good way to see those changes is through the git integration. If this wasn't in a git repository, I'd be more screwed (P_7).

In order to limit problems related to code impact, some participants carefully orchestrated the order in which they applied refactoring tools to the call graph, starting on what they referred to as “leaf nodes” or nodes with few callers. In this way, they controlled the impact of the tool because they made sure that there were only ever a few test callers to propagate changes to. P_{16} employed this workflow throughout the entire class, whereas others like P_4, P_7, P_9, and P_{10} realized only partway through. In this case, the order in which the tools were applied impacted the cost associated with using them.

Participants experienced code layout as relevant in Tasks 1 and 3. In the first task, P_{14} and P_{19} used the `move`refactoring to move one method and indicated that using the tool was not worth it. They continued the task by manually moving code. P_{14} commented that the layout of the code (i.e., the methods being declared after one another) made the manual approach easier: “It's nice that these are all together. Then I can just wholesale them over. If they were scattered throughout then maybe it would be worth the extra steps.” Similarly, in Task 3, many of the participants (e.g., P_8, P_{11}, P_{12}, P_{14}), who manually inlined methods to their wrapper method, observed the convenience of the wrapper being located just above each method declaration, allowing them to make only a single text selection. Even P_9 and P_{14}, who initially used the `inline-method` on this task, eventually reverted to this workflow.

Among the participants who successfully used refactoring tools throughout the entirety of Task 3, we observed an interesting workflow. Two participants used “problem” views or “preview” views to learn about the code before changing it *manually* by acting on the information these views presented. In Task 3, P_5 used `safe-delete` and invoked it twice for each of the eight methods that had to be changed: the first time, they used the preview view to navigate to code that would be impacted by the change and updating some of these elements manually before invoking it again and applying the refactoring. P_8 used a similar workflow, where they invoked `change-signature` and inspected the “problem” view before canceling the refactoring and locating the problematic elements in the code, changing them, and reapplying the refactoring. This also provides empirical evidence for the claim posed by Fleming et al. that a developer who is refactoring needs to forage for *cues* for how the refactoring should be performed and that refactoring tools help present such cues.[49]

In summary, we observed that participants' success with tools was impacted by experiences with the tool's impact on test code, difficulties understanding the tool's impact on code, and code layout. The participants who successfully used tools were careful with the order in which they applied refactorings or used the information presented by the tools to organize their manual changes. To our knowledge, there is no earlier work that brings up this distinction of test code from functional code in refactoring tools, nor reports of developer reliance on diff tools like git to learn about a tool's impact after it is applied. Some earlier works have indicated that developers avoid using refactoring tools due to fear of merge conflicts or code ownership.[32] The following observations are, to our knowledge, novel: that participants organize refactoring applications along the call graph, use information from the tool to organize manual changes, and that code layout impacts the choice to use or not use tools are.

5.3.4 | Experiences and strategies

We looked into whether participant strategies impacted their experience of tools. Figure 7 shows the drop-off trend relative to their strategies. We see that a drop-off happens more often when the participant uses a **Local** strategy and less often when they use a **Structure** strategy. This shows that participants who used a **Structure** approach are not the only ones who are aware of tools, but awareness led to use more often in this group. We speculate that participants who employed the other strategies might more often encounter experiences with the tools that lead them to avoid using them.

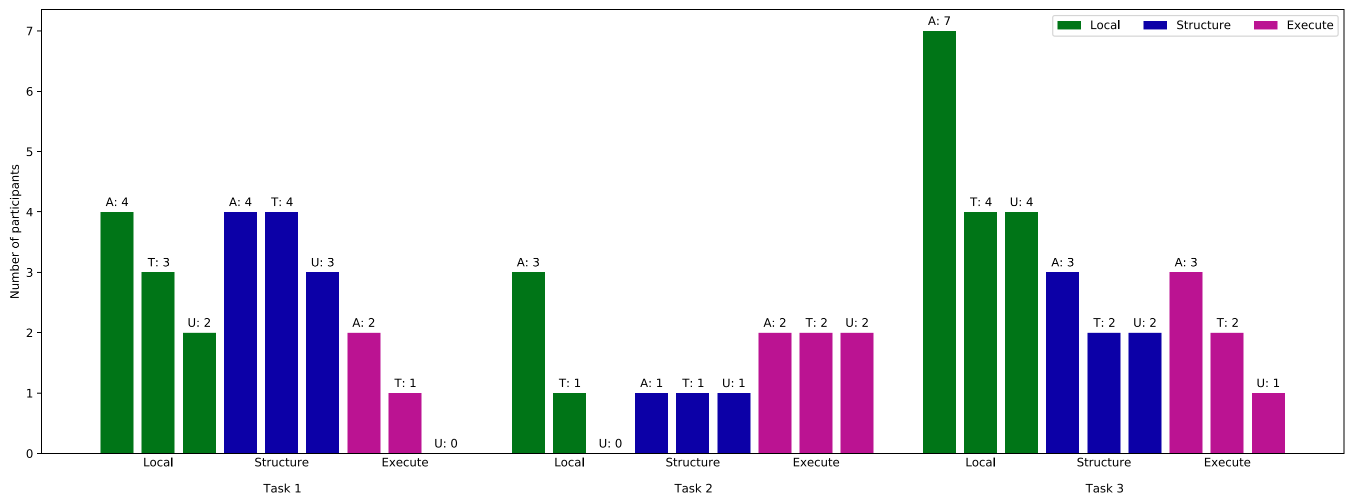

FIGURE 7 In each of the three tasks, developers from all strategies indicated awareness of tools. The leftmost bar of each color indicates, for that task, the number of participants who employed that strategy on the task and expressed awareness of tools. The middle bar of each color indicates how many participants tried a tool, and the rightmost one (if any) indicates how many participants from that strategy *used* a tool on that task

In Task 2, participants who employed a **Local** strategy were prone to removing the method that implemented the functionality *before* navigating to callers. The workflow that these participants used blocked them from using the refactoring tool unless they restarted the task. Once these participants were in a position to invoke `inline-method` (after they realized there was a caller), the tool no longer supports the refactoring because the method has been deleted.

In fact, across all tasks, the only participants that ever invoked `inline-method` used using either a **Structure** or **Execute** strategy. We speculate that participants who employ a **Local** strategy did not look for this tool because its name does not represent a local change. In comparison, `change-signature` or `safe-delete` has names that might be more aligned with these participants' intent.

Among participants who encountered problems with unintended code changes in test code in Task 3, the ones using an **Execute** strategy expressed a lower tolerance for this: P_9 and P_{10} restarted the task to find a different approach, and P_{16} carefully ordered the applications along the call graph. These participants' strategies were focused on keeping the tests running and isolating code changes. In comparison, many of the participants who approached the tasks with a **Local** or **Structure** strategy did not inspect test code until the end of the task at which point they realized the tools' impact.

Finally, the two participants who exhibited the interesting refactoring tool workflow in Task 3, P_5 and P_8, were both utilizing a **Local** strategy. We observe that these participants were able to have information from the "preview" and "problem" views guide their changes in a way that we find comparable to participants who had their changes guided by compiler errors: rather than considering the information that these views presented as a hindrance to applying the tools, these two participants gathered information that they used to guide their subsequent (manual) actions. While other participants sometimes inspected these views before taking other actions, we saw no other participants employ this workflow throughout the entire task.

5.4 | Summary of results

Here, the novel findings from this study are briefly summarized.

Participants approached each task using one out of three strategies: **Local**, **Structure**, and **Execute**. The choice of strategy was dependent on the individual not the task. Refactoring tools were used across all strategies employed by participants but most commonly seen when participants approached the task with a **Structure** strategy despite participants with all strategies showing awareness of tools. All participants had one or more opportunities to use refactoring tools as part of their workflows on all of the three tasks, but out of these 51 workflows, only 17 (33%) included refactoring tool use, despite participants expressing awareness of tools in 32 (63%) of them. This degree of tool use is in line with previous findings (Section 2).

The following experiences impacted the participants' choice or ability to use refactoring tools. Participants who both used and did not use tools experienced lacking control of, and understanding of, the impact of the tool. This occurred particularly frequently when the tool updated test callers in Task 3. Participants navigated this problem by relying on git diff or by carefully ordering the applications of the tool according to the call

graph. Several participants experienced this reliance on git diff as bothersome yet unavoidable. Participants who were *unwilling* to use tools experienced that the tool deprived them of knowledge about the code that they could use later in their workflow. Participants who were *unable* to use tools experienced problems invoking them due to incorrectly assuming their code selection was passed to the tool as a refactoring argument.

When participants used refactoring tools, we observed novel workflows in which participants used the "preview" view or "problem" view of refactoring tools to organize manual changes among the participants using a **Local** strategy. We found several examples of participant strategies impacting their experiences with refactoring tools.

Finally, participants who decided to use or not use refactoring tools performed a *cost–benefit analysis* on the fly in which they compared the perceived cost associated with using a tool with the perceived benefit that the tool posed. Examples of factors that impacted this analysis are the aforementioned experiences and code layout.

6 | THREATS TO VALIDITY

There are various threats to the validity of our study and the finding and conclusions presented in this paper.

6.1 | Construct validity

The construct validity of our study is impacted by various aspects of the study setup. The participants in the study knew the study was about refactoring tools and were thus primed and more likely to try to use the tools than in their normal workflow. We were willing to accept this threat to enable more occurrences of refactoring tool use. Participants reported making choices of proceeding manually on tasks, so we believe the impact of this threat on our study results is reduced.

In some cases, participants were also not working with their normal development environment and tools, which may have impacted their approach. We mitigated this threat by answering questions they had about the tools we provided. Finally, participants were working on unfamiliar tasks on an unfamiliar system and were being asked to think aloud as they worked. We accept these threats to enable detailed observation and comparison of different individuals on the tasks.

6.2 | Internal validity

The internal validity of our study could have been affected by our choice to use an observational study in which the experimenter interacted with the participants. These interactions might have impacted participants' actions and responses, as can their awareness of being observed. We chose this study design because we determined that it was necessary to observe and interact with developers to ensure that they provided complete descriptions of their process and to pick up on topics to pursue in interviews. We took steps to mitigate these risks by remaining neutral during sessions not bringing up refactoring names before participants do and delaying certain questions until after all tasks were done. We accept that this threat cannot be fully mitigated.

6.3 | External validity

Our study also has threats to the external validity of the findings. Our study included only three tasks and only 17 participants. We chose the tasks to replicate ones that had been committed to open source systems, thus improving the realism of the tasks. Six participants made unprompted comments about having encountered similar tasks to Task 3 in their own work, which indicates that this threat is mitigated. We recruited participants who represent our target demographic in terms of experience levels and gender distribution. With these steps, we believe our findings are sufficiently valid to generate hypotheses to drive considerations for tool improvements and for the role of refactoring in change tasks.

6.4 | Conclusion validity

In an observational study like this one, a risk to conclusion validity is to inadequately analyze or understand the available data. Our labeling of transcripts used in the analysis may be biased by our interpretations of developer statements and actions. The conclusions we draw from the analyzed transcripts may be biased by our priming from reviews of the previous literature. We have attempted to mitigate the effects of these biases

by detailed our coding method and by having more than one individual involved in the analysis. We also included in the study design an interview segment where participants were asked to elaborate on or explain points or actions that the experimenter noted during their work on the tasks.

Future observations of industrial developers at work or case studies could enable investigation of our findings in situations that reduce the threats associated with this study design. Moreover, these findings could be validated by means on an online survey in a larger sample of the demographic.

7 | DISCUSSION

The results of our study suggest several ways to improve refactoring tools and to better investigate the disuse of these tools. Our results also call into question whether refactoring tools, as currently formulated, are the most effective way to aid developers in facing refactoring steps that can be automated as part of their work.

7.1 | Improving refactoring tools

We saw participants in our study struggle to use refactoring tools because of problems *controlling* the impact of the tool and being deprived of *knowledge* about the code and about the changes that the tool made.

7.1.1 | Controlling the impact of refactorings

We were surprised to observe how many participants used **Execute** strategies to perform the tasks. The participants using this strategy were focused on keeping the tests running, and in many cases, this led them to elegant solutions. In particular, after isolating the functional change of interest to the system, these participants could perform the remaining changes, such as inlining the methods (Task 2) or removing the parameter (Task 3) as a *pure* refactoring akin to removing dead code.

We were also surprised to see the participants using an **Execute** strategy frequently avoided using refactoring tools or stopped using the tool despite initially trying it. Although the **Execute**strategy could help enable the use of refactoring tools, developers who take this approach are hindered in their use because the tools do not distinguish between functional and test code.

We are reminded of this description of refactorings from Opdyke's thesis,[7] §.4.1:

> Imagine that a circle is drawn around the parts of a program affected by a refactoring. (…) the key idea is that the results (including side effects) of operations invoked and references made from outside the circle do not change, as viewed from outside the circle.

If our participants could have drawn a circle to exclude the tests and then could have invoked a refactoring tool on the inner part of the circle, we believe they may have been more inclined to use a refactoring tool.

Although tools frequently offer preview windows that communicate the refactoring's impact on the program, our study illustrates the lack of fine-grained control of this impact. We saw instead that the participants who attempted to gain such control resorted to first understanding the call graph, then applying the tool to callers in order. Toolmakers may enable such control by, for example, integrating into the tool a richer program model. Such a model could allow for defining refactorings that not only change the program in occordance with the target programming language but could also take into account, for example, the separation between test code and production code, different levels of security or ownership[32] or offer user-defined control of impact areas.

7.1.2 | Understanding the impact of refactorings

When using a refactoring tool, participants in our study frequently wondered what in the program the tool had changed. Despite using the preview capability of the refactoring tool, after the tool was run and its transformation applied, every participant questioned at some point what had happened, asking: “What did the tool do?” or “How did this code change?” These participants relied on undoing and redoing the changes made by the tool or referring to git to understand how the use of a refactoring tool had changed their code. Several described this lack of knowledge about the impact a tool had on their code as a reason they did not *trust*tools.

Previous studies have linked the inability to understand the tool's impact to a lack of preview windows.[6,13] Clearly, the preview capabilities of refactoring tools as provided in IntelliJ do not suffice for developers. This may be explained by the tool impacting code that they were not yet

attuned to think about. In several cases, the participants in our study asked these questions at a later stage in their work, such as when they moved from the functional code file to the tests. At the time they previewed the refactoring, they were not yet focused on the test code. In addition to a preview capability, we suggest refactoring tool designers consider a comprehensive and undoable record of changes that is accessible after the fact to help developers understand the changes the refactoring tools made to the code.

7.2 | Investigating refactoring tool disuse

Studies of refactoring disuse have sometimes taken an approach of mining potential refactorings from version histories of code.[3] Such studies consider how prevalent the different refactorings may be[6,13] and have attempted to learn why an automated approach may not have been used by interviewing the developers who committed the code.[5] In essence, these studies take a backward approach to investigating refactoring tool disuse: let us find places where refactoring may have happened and attempt to understand how tools failed or how they could have helped. The observational study we report on in this paper takes a forward approach to investigating refactoring tool disuse by considering how the approaches used by developers enable or remove the possibility of refactoring tool use.

We based the tasks we used in our study on commits from open-source systems for which refactoring mining tools indicated a particular refactoring had occurred. For Task 3, the RefactoringMiner tools reported the use of `inline-method`. Interestingly, only three developers in our study applied this refactoring, whereas the remaining participants used different refactoring operations. This is not a limitation of the tool: the refactorings were correctly identified in the source code change; instead, this discrepancy should be understood to illustrate the limitations of both the forward and backward approaches to studying refactoring tool (dis)use. The backward approach suggests a refactoring that may have been impossible for that developer to use; that developer's approach may not have enabled them to use a refactoring tool. Our forward approach is limited in the breadth of approaches that the developers participating in our study considered.

We hope that the observations from our study, which show the importance and impact of the context of how a developer work, may help broaden the study of refactoring use beyond the actions of a developer close in time to when a refactoring tool is used.

7.3 | Bridging the gap between developer workflow and refactoring tools

In our study, we saw evidence that the way a developer thinks about the task and the plan they make to perform the task impacts the universe of tools that might be usable as they work.

As a simple example, consider Task 1 in which participants were asked to organize test methods. Regardless of the strategy a participant used, they all solved Task 1 in one out of two ways: (1) either by *creating* a new class and *moving* methods into the class or (2) by *duplicating* code (e.g., copy-pasting an entire file) and *reducing* each file to contain only the appropriate code.

We saw the universe of tools that the participant might consider to help perform the task impacted by which of the ways they proceeded to work on the task. Some participants using the *create-move* approach considered a `move` refactoring tool for moving methods into their new class. We conjecture it was easier for the participants to map the way they were working to a tool that was similarly named and seemed to match the actions they wished to perform.

Other participants struggled with mapping the description of their approach—their cognitive model—to the refactoring tools available. For example, P_2 did attempt to use a tool after a prompt but was unable to invoke the correct refactoring: ".. what am I moving, am I moving the entire class? That's not what I want. Extract? Oh that didn't work. Extract .. Method? Well that's not what I want either." They reverted to their manual approach.

Developers who approached the task using *duplicate-reduce* immediately also reduced the opportunities to employ a refactoring tool. Neither the operations they wished to perform, nor the way they were thinking about the problem, lent themselves to refactoring operations.

A similar reduction of opportunity occurred for `inline-method` in Task 2. Participants who approached this task by deleting the method an inspecting the resulting compiler errors would learn about the caller too late. When these participants were able to recognize that they should apply `inline-method to the caller`, they had already deleted the method definition to provoke compiler errors, and consequently, there was no longer any method definition to inline.

When participants could easily match their intent to a tool description, they also had more success in using refactoring tools. For example, during Task 3, several participants were able to select `change-signature` without ever having used it before. When prompted about why they picked this tool, P_4 said, "That seemed like the appropriate thing because that is what I was doing." In this task, participants who were able to have the tool invocation capture their intent could also utilize the information it provided them in their manual changes, as was observed with P_5 and P_8.

A challenge in improving refactoring tool use is to bridge the gap between the developer's approach and intent to the work they are performing and the tools available. We observed that many participants would have needed to deviate from their workflows in order to encounter

the code change that was supported by the tool. The gulf between tools and workflows may be larger than has been previously considered for refactoring tools.

Future studies should consider developer approaches to tasks in more detail. Perhaps it is possible to guide a developer to a strategy and workflow that enables the use of refactoring tools through suggestions. In other cases, such as where code layout enables easier manual changes, developers may benefit from help with a cost–benefit analysis to evaluate whether using a refactoring tool is worthwhile. Alternatively, there may be other tools more useful to developers, such as ones that help assess the impact, risk, or consequence of changes to the code. Such a tool may support manual changes a developer is performing that may or may not be similar to refactoring operations.

8 | CONCLUSION

The formative study we conducted involving 17 experienced developers performing three change tasks on a nontrivial system provides novel observations about factors that affect refactoring tool use and disuse. We observed the participants in our study used a variety of strategies (**Local**, **Structure**, and **Execute**) to work on the same task and that refactoring tools were used within all three of the strategies. We also observed the difficulties participants faced in finding the right-sized transformation to consider using a refactoring tool: some transformations were small and easier to perform manually; others were large, making participants wary of trusting refactoring tools.

Given the large literature on refactoring tool disuse, we were surprised to learn new ways in which refactoring tools fail for users. Specifically, we learned that while the preview capability that shows the impact of a refactoring operation before it is run is useful, our participants also wanted a summary of the impact of the tool after the operation had changed the code. Our participants also desired more control before running the tool; they wished to control the impact it had on their code by limiting the scope of the program on which it would apply. In particular, they desired to limit the impact of the refactoring tool on the program's tests.

With this observational study, we have opened up new avenues for further exploration of refactoring tools and other tools. We have provided observations that can benefit both toolmakers in designing better tools and practitioners in better utilizing existing tools. By demonstrating the impact of how a developer works on whether tools are even applicable, we suggest that there are rich avenues to consider of how to bridge the gap between developers and refactoring tools.

CONFLICT OF INTEREST

The authors declare that they have no conflicts of interest.

DATA AVAILABILITY STATEMENT

The data that support the findings of this study are openly available in Dataverse at https://doi.org/10.18710/VTTNXM

ORCID

Anna M. Eilertsen https://orcid.org/0000-0002-8586-8460

REFERENCES

1. Hindle A, German DM, Holt R. What do large commits tell us? A taxonomical study of large commits. In: Proceedings of the 2008 international working conference on mining software repositories; 2008:99-108.
2. Tufano M, Bavota G, Poshyvanyk D, Penta MD, Oliveto R, Lucia AD. An empirical study on developer-related factors characterizing fix-inducing commits. *J Softw Evol Process*. 2017;29(1):e1797.
3. Tsantalis N, Mansouri M, Eshkevari LM, Mazinanian D, Dig D. Accurate and efficient refactoring detection in commit history. In: Proceedings of the 40th international conference on software engineering. Association for Computing Machinery; 2018:483-494.
4. Dig D, Johnson R. How do apis evolve? A story of refactoring. *J Softw Maint Evol Res Pract*. 2006;18(2):83-107.
5. Silva D, Tsantalis N, Valente MT. Why we refactor? Confessions of github contributors. In: Proceedings of the 2016 24th acm sigsoft international symposium on foundations of software engineering, FSE 2016. New York, NY, USA: ACM; 2016:858-870. http://doi.acm.org/10.1145/2950290.2950305
6. Murphy-Hill E, Parnin C, Black AP. How we refactor, and how we know it. In: Proceedings of the 31st International Conference on Software Engineering, ser. ICSE '09. Washington, DC, USA: IEEE Computer Society; 2009:287-297. https://doi.org/10.1109/ICSE.2009.5070529
7. Opdyke WF. Refactoring object-oriented frameworks. *Ph.D. Thesis*: University of Illinois at Urbana-Champaign; 1992.
8. Fowler M. *Refactoring: improving the design of existing code*: Addison-Wesley; 2018.
9. Paixão M, Uchôa A, Bibiano AC, et al. Behind the intents: an in-depth empirical study on software refactoring in modern code review. 17th MSR; 2020.
10. Palomba F, Zaidman A, Oliveto R, De Lucia A. An exploratory study on the relationship between changes and refactoring. In: 2017 ieee/acm 25th international conference on program comprehension (icpc) IEEE; 2017:176-185.
11. Eclipse refactoring documentation. https://help.eclipse.org/2020-12/index.jsp?topic=%2Forg.eclipse.jdt.doc.user%2Freference%2Fref-mnu-refactor.htm, verified 21/12/2020, [Online: accessed 01-January-2021].

12. Visual studio refactoring documentation. https://docs.microsoft.com/en-us/visualstudio/ide/refactoring-in-visual-studio?view=vs-2019, note = “[Online: accessed 01-January-2021]”.
13. Vakilian M, Chen N, Negara S, Rajkumar BA, Bailey BP, Johnson RE. Use, disuse, and misuse of automated refactorings. In: 2012 34th international conference on software engineering (icse); 2012:233-243.
14. Sharma T, Suryanarayana G, Samarthyam G. Challenges to and solutions for refactoring adoption: an industrial perspective. *IEEE Softw*. 2015;32(6): 44-51.
15. Foster SR, Griswold WG, Lerner S. Witchdoctor: Ide support for real-time auto-completion of refactorings. In: 2012 34th international conference on software engineering (icse); 2012:222-232.
16. Ge X, DuBose QL, Murphy-Hill E. Reconciling manual and automatic refactoring. In: Proceedings of the 34th international conference on software engineering, ICSE '12. Piscataway, NJ, USA: IEEE Press; 2012:211-221. http://dl.acm.org/citation.cfm?id=2337223.2337249
17. Murphy-Hill E, Black A. Breaking the barriers to successful refactoring. In: 2008 acm/ieee 30th international conference on software engineering; 2008:421-430.
18. Schaefer M, de Moor O. Specifying and implementing refactorings. In: Proceedings of the acm international conference on object oriented programming systems languages and applications, OOPSLA '10. New York, NY, USA: ACM; 2010:286-301. http://doi.acm.org/10.1145/1869459.1869485
19. Mongiovi M, Gheyi R, Soares G, Ribeiro M, Borba P, Teixeira L. Detecting overly strong preconditions in refactoring engines. *IEEE Trans Softw Eng*. 2018;44(5):429-452.
20. Vakilian M, Chen N, Moghaddam RZ, Negara S, Johnson RE. A compositional paradigm of automating refactorings. In: European conference on object-oriented programming Springer; 2013:527-551.
21. Abid C, Alizadeh V, Kessentini M, Do Nascimento Ferreira T, Dig D. 30 years of software refactoring research:a systematic literature review; 2020.
22. Bannwart F, Müller P. Changing programs correctly: refactoring with specifications. In: International symposium on formal methods Springer; 2006: 492-507.
23. Steimann F. Constraint-based refactoring. *ACM Trans Program Lang Syst*. 2018;40(1):2:1-2:40. http://doi.acm.org/10.1145/3156016
24. Reichenbach C, Coughlin D, Diwan A. Program metamorphosis. In: Proceedings of the 23rd european conference on ecoop 2009 — object-oriented programming, Genoa. Berlin, Heidelberg: Springer-Verlag; 2009:394-418. https://doi.org/10.1007/978-3-642-03013-0_18
25. Eilertsen AM, Bagge AH, Stolz V. Safer refactorings. In: International symposium on leveraging applications of formal methods Springer; 2016: 517-531.
26. Mongiovi M. Scaling testing of refactoring engines. In: 2016 ieee/acm 38th international conference on software engineering companion (icse-c); 2016:674-676.
27. Kim J, Batory D, Dig D, Azanza M. Improving refactoring speed by 10x. In: 2016 ieee/acm 38th international conference on software engineering (icse) IEEE; 2016:1145-1156.
28. Lee YY, Chen N, Johnson RE. Drag-and-drop refactoring: intuitive and efficient program transformation. In: Proceedings of the 2013 international conference on software engineering, ICSE '13. Piscataway, NJ, USA: IEEE Press; 2013: 23-32. http://dl.acm.org/citation.cfm?id=2486788.2486792
29. Murphy-Hill E, Ayazifar M, Black AP. Restructuring software with gestures. In: 2011 ieee symposium on visual languages and human-centric computing (vl/hcc); 2011:165-172.
30. Steimann F, von Pilgrim J. Refactorings without names. In: Proceedings of the 27th ieee/acm international conference on automated software engineering, ASE 2012. New York, NY, USA: ACM; 2012: 290-293. http://doi.acm.org/10.1145/2351676.2351726
31. Campbell D, Miller M. Designing refactoring tools for developers. In: Proceedings of the 2nd workshop on refactoring tools, WRT '08. New York, NY, USA: ACM; 2008: 9:1-9:2. http://doi.acm.org/10.1145/1636642.1636651
32. Kim M, Zimmermann T, Nagappan N. A field study of refactoring challenges and benefits. In: Proceedings of the acm sigsoft 20th international symposium on the foundations of software engineering, FSE '12. New York, NY, USA: ACM; 2012: 50:1-50:11. http://doi.acm.org/10.1145/2393596.2393655
33. Liu W, Liu H. Major motivations for extract method refactorings: analysis based on interviews and change histories. *Front Comput Sci*. 2016;10(4): 644-656. https://doi.org/10.1007/s11704-016-5131-4
34. Leppänen M, Mäkinen S, Lahtinen S, Sievi-Korte O, Tuovinen A-P, Männistö T. Refactoring-a shot in the dark? *IEEE Softw*. 2015;32(6):62-70.
35. Brant J, Steimann F. Refactoring tools are trustworthy enough and trust must be earned. *IEEE Softw*. 2015;32(6):80-83.
36. Eilertsen AM. Predictable, flexible or correct: trading off refactoring design choices. In: ICSE '20: 42nd international conference on software engineering, workshops, seoul, republic of korea, 27 june - 19 july, 2020. ACM; 2020:330-333. https://doi.org/10.1145/3387940.3392185
37. Eilertsen AM, Murphy GC. The usability (or not) of refactoring tools. In: 2021 ieee 28th international conference on software analysis, evolution and reengineering (saner) IEEE; 2021.
38. Ergonomics of human–system interaction part 11: Usability: Definitions and concepts iso 9241-11: 2018 (en). https://www.iso.org/obp/ui/#iso:std:iso:9241:-11:en, (Accessed September 10th, 2020).
39. Apache commons-lang. https://github.com/apache.org/proper/commons-lang, [Online: accessed 25-August-2020]; 2020.
40. Quasar Commit 56d4b99. https://github.com/puniverse/quasar/commit/56d4b999e8be70be237049708f019c278c356e71, [Online: accessed 24-July-2020].
41. Maven. https://maven.apache.org/, [Online: accessed 28-August-2020].
42. Apache commons-lang commit 0223a4d. https://github.com/apache/commons-lang/commit/0223a4d4cd127a1e209a04d8e1eff3296c0ed8c1, [Online: accessed 24-July-2020].
43. Apache commons-lang commit 3ce7f9e. [Online: accessed 24-July-2020].
44. Apache commons-lang commit 3ce7f9e pull request. https://github.com/apache/commons-lang/pull/221, [Online: accessed 24-July-2020].
45. Intellij refactoring documentation. https://www.jetbrains.com/help/idea/refactoring-source-code.html, [Online: accessed 01-January-2021].
46. Ko AJ, Myers BA, Coblenz MJ, Aung HH. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Trans Software Eng*. 2006;32(12):971-987.
47. Maalej W, Tiarks R, Röhm T, Koschke R. On the comprehension of program comprehension. In: Software engineering & management 2015, multikonferenz der gi-fachbereiche softwaretechnik (SWT) und wirtschaftsinformatik (wi), FA wi-maw, 17. märz - 20. märz 2015, dresden, germany Aßmann U, Demuth B, Spitta T, Püschel G, Kaiser R, eds., LNI, vol. P-239. GI; 2015:65-68.

48. Negara S, Chen N, Vakilian M, Johnson RE, Dig D. A comparative study of manual and automated refactorings. In: European conference on object-oriented programming Springer; 2013:552-576.
49. Fleming SD, Scaffidi C, Piorkowski D, et al. An information foraging theory perspective on tools for debugging, refactoring, and reuse tasks. *ACM Trans Softw Eng Methodol (TOSEM)*. 2013;22(2):1-41.
50. Eilertsen AM, Murphy GC. Data Package. https://doi.org/10.18710/VTTNXM

How to cite this article: Eilertsen AM, Murphy GC. A study of refactorings during software change tasks. *J Softw Evol Proc*. 2021;e2378. doi:10.1002/smr.2378

APPENDIX

This appendix contains brief descriptions of the refactoring operations that are referred to in this paper. For a complete list of the refactoring tools that participants had available in the study, consult the IntelliJ documentation.[45] For a general introduction to refactorings, we suggest Fowler. [8]

`safe-delete`

removes an element (e.g., method or parameter) if there are no references to the element.

`extract-class`

takes a number of selected members (e.g., methods or fields) and declares a new class with these members. It also removes them from the original location and updates all references to them.

`move-method`

moves a method declaration to another location, such as another class. It also updates any callers to the method.

`inline-method`

replaces calls to a method with the method's body and removes the declaration of the method that was inlined.

`change-signature`

allows for editing the signature of a method, for example, by removing unused parameters. It automatically updates callers.

`remove-parameter`

takes an unused parameter in a method declaration and removes it from the declaration and from all callers.

`extract-method`

refactoring takes a sequence of statements, creates a new method with the statements as its body, and replaces the original sequence with a call to this method.