The Usability (or Not) of Refactoring Tools

Anna Maria Eilertsen
Department of Informatics
University of Bergen
Bergen, Norway
anna.eilertsen@uib.no

Gail C. Murphy
Department of Computer Science
University of British Columbia
Vancouver, Canada
murphy@cs.ubc.ca

***Abstract*—Although software developers typically have access to numerous refactoring tools, most developers avoid using these tools despite their benefits. Researchers have identified many reasons for the disuse of refactoring tools, including a lack of awareness by the developers, a lack of predictability of the tools, and a lack of need for the tools. In this paper, we build on this earlier work and employ the ISO 9241-11 definition of usability to develop a theory of usability for refactoring tools. We investigate existing refactoring tools using this theory by analyzing how 17 developers experience refactoring tools in three software change tasks we asked them to perform. We analyze qualitatively the resulting interview transcripts based on our theory and report on a number of observations that can inform tool designers interested in improving the usability of refactoring tools. For instance, we found a desire for developers to guide how a refactoring tool changes the code and a need for refactoring tools to describe changes made to developers. Refactoring tools are currently expected to preserve program behavior. These observations indicate that it may be necessary to give developers more control over this property, including the ability to relax it, for the tools to be usable; that is, for the tools to be effective, efficient and satisfying for the developer to employ.**

***Keywords*-automation, tool usability, software evolution**

I. Introduction

During software evolution and maintenance, developers frequently apply *refactoring* operations to source code [1], [2] for the purpose of improving its quality [3] or preparing and completing functional code changes [4], [5]. *Refactoring tools* automate the application of refactoring operations and reduce development costs by automatically performing code changes that are time-consuming and error-prone for developers to apply manually.

As a simple example, a developer may change the name of a method in order to better represent its functionality. If the program should behave as before, all references to the method throughout the program must be updated to use the new name. This `rename` refactoring operation, and many others, such as extracting, moving and inlining program elements, occur during software change activities [1]–[4], [6] and enjoy extensive automated support in most mainstream development environments, such as Eclipse [7] and IntelliJ [8].

Developers' refactoring activities have been studied for over three decades and much effort has been made to automate refactoring support [9]. While some automated operations, like the aforementioned `rename`, have been readily embraced by software developers, most refactoring operations are applied manually [1], [4], [10], [11] and refactoring tools are, in fact, *disused* [2].

Empirical studies indicate that developers disuse refactoring tools that are present in their programming environments—even when they are aware of them—due to usability issues [1], [2], [4], [10], [12]. Researchers have proposed various approaches to the reported usability problems [13]–[20], most of which are aimed at improving singular usability aspects such as speed [21], selecting code [12] or choosing the right refactoring operation [18]–[20]. There is no comprehensive approach to refactoring tool usability nor do we understand what causes a developer to find a refactoring tool trustworthy or predictable [22], [23].

In this paper, we seek to provide a framework to help guide refactoring tool designers in creating refactoring tools that developers choose to use. We derive a theory of the usability of refactoring tools (Section III) based on the ISO 9241-11 definition of usability [24]. This new theory differs from existing theories (Section II) in two ways. First, it considers a more common context of use of the tools, namely to prepare or complete functional changes to a system. Second, it considers what developers seek as an experience in using the tools.

We then use the newly posited theory to study the use of refactoring tools by 17 software developers attempting three functional software change tasks on a non-trivial software system. As the participating software developers performed the tasks, we captured their think-aloud [25] descriptions. We also conducted semi-structured interviews with the participants after each task and at the end of a study session.

We analyze the transcripts captured from the 32 hours of study sessions by employing the introduced theory (Section III) and findings from previous empirical studies of refactoring tools (Section II). We use a coding approach to investigate the participants' comments according to five usability factors—*effectiveness*, *efficiency*, *satisfaction*, *predictability* and *trust*, finding that the participants' experience with refactoring tools is dominated by their perceptions of the predictability of the tools. Next, we perform card sorts [26] of the comments associated with each of these factors: i.e., we organize the comments into groups until meaningful categories emerge. A synthesis of the results indicates a need

This work is supported by the Research Council of Norway under grant number 250683 *(Co-Evo)*.

for refactoring tools to better communicate their capabilities and the changes they make to code to the developers that use the tools. The synthesis also indicates that developers need a means to better assess up-front the costs and benefits of using a tool and require the ability to guide how a tool changes code. Based on this synthesis, we describe refinements to our theory and suggest implications for refactoring tool designers (Section VII), such as enabling a step-through workflow similar to that in debugging tools and having the tools provide a summary of changes made to the code after they are applied.

This paper makes three contributions:

- it presents and refines a theory of refactoring tool usability,
- it uses the theory to investigate the experiences of 17 practicing software developers attempting three software change tasks known to be amenable for the use of refactoring tools, and
- it presents a set of recommendations for refactoring tool designers to consider when building refactoring tools.

The refined theory states that:

> *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.*

We begin by reviewing related work on theories about, and studies of, refactoring tool use (Section II). We then apply the ISO 9241-11 to derive a theory of usability for refactoring tools (Section III) and employ the theory as a perspective to analyze how developers in a study experience refactoring tools (Section IV). Next, we describe the analysis we use to consider usability factors for refactoring tools (Section V) and present the results of the study (Section VI). We discuss a revised theory and implications for refactoring tool designers as well as threats (Section VII) before summarizing the paper (Section VIII).

II. Related Work

We focus our descriptions of the extensive existing research on refactoring tools on: 1) existing theories of the process of using refactoring tools, 2) empirical studies focused on the usability of refactoring tools, and 3) work that explores similar questions.

Mens and Tourwe [27] is the only work of which we are aware that presents a theory of the process of using refactoring tools: 1) determine where a refactoring should occur in software, 2) determine which refactoring operation should be applied, 3) ensure the operation will preserve program behavior, 4) apply the refactoring operation and assess the quality improvement, and 5) restore consistency between software artifacts. This theory is predicated on refactoring operations being applied only to improve quality attributes of software. However, investigations into refactoring use have shown that refactoring operations are often performed as part of functional changes [4], [5]; our theory incorporates this usage pattern.

Murphy-Hill proposes a model of refactoring tool use that extends this theory by including steps in which the user iteratively apply refactorings and interpret errors and results [28]. Our study is designed to enable this form of use of refactoring tools. Their model describe the steps that a user perform when interacting with the tool. In contrast, our theory describes usability of the tool.

To better understand how developers use refactoring tools, a number of researchers have performed a range of studies. Across these studies, a common finding is that refactoring tools are disused: Murphy-Hill et al. find that 90% of refactorings are performed manually [1], Kim et al. find over 50% to be manual [10], Negara et al. find over 50% to be manual [11], Vakilian et al. find that over half of their participants perform refactorings manually sometimes [2], and Silva et al. find only 38% of the refactorings they studied to be performed with a tool [4].

To better understand this disuse and improve utilization of refactoring tools, several studies have investigated usability issues that deter developers from using them. Murphy-Hill et al. [1] combined Eclipse user data, open-source repositories and survey results from a small set of developers to understand refactoring tools in action. They found that most refactoring operations are interleaved with other code changes to avoid degrading the code's quality (*floss refactoring*) while refactorings that are performed in isolation to restore or improve software quality (*root-canal refactoring*) are more rare. They reported that developers avoid using tools due to lack of awareness that their code changes could be automated and a lack of opportunity to use refactoring tools. They also report that developers avoid tools due to lack of trust, tools disrupting their workflow or not properly presenting touch points—code areas that are impacted by the tool.

In another study [12], Murphy-Hill and Black instructed participants to perform the `extract-method` operation with a tool. They reported that participants did not understand error messages and made incorrect code selections. They presented recommendations for improving code selection and the presentation of error messages and warnings in the `extract-method` tool.

Vakilian et al. [2] combined IDE monitoring and interviews to investigate usability issues in refactoring tools. They found a number of factors that impacted the use of automated refactoring tools: the perception that the tool is unnecessary or unneeded; lack of awareness (similar to [1]); problems recalling names of refactoring operation; lack of trust in the tools; difficulty in predicting the outcome of tools; and usability issues with configuration windows. In later work, the authors address the observation that developers prefer to automate simple refactorings and disuse complex refactorings, proposing that tools should support simple, composable refactoring operations [17].

Silva et al. [4] monitored open-source projects for commits that contain refactorings. Upon detecting a refactoring commit, an email was automatically sent to the author of the commit asking why they applied the refactoring and if they applied it using automated refactoring support. The most frequently reported reason for disuse was lack of trust in tools for complex refactorings followed by claims that tools are unnecessary due to low complexity. Other factors included lack of tool support, lack of familiarity with tool support and lack of awareness.

While many usability factors are well-described, such as lack of awareness and tool support, there are two factors that arise across these studies, yet are not well understood: trust and predictability. Campbell and Miller [29] use the term trust to describe developers' uncertainty of how a tool will change their program or a fear that it will "mangle their code". Brant and Steimann [22] discussed whether tool correctness is necessary for developers to trust and use a tool and do not reach a conclusion. Oliveira et al. [23] conducted a survey where developers chose from presented outputs which they expect from specific refactoring operationsand found that respondents rarely agree about what to expect, nor do their expectations align with the behavior of the refactoring tools. We use *trust* and *predictability* as *sensitizing concepts* [30] for interviews in the empirical study we conduct.

Mealy et al. [31] also use the ISO 9241-11 definition to frame the context in which refactoring tools operate. They use it to contextualize usability design guidelines gathered from the literature and propose 81 usability design guidelines for refactoring tools, such as automate code-smell identification. They analyze four refactoring tools using the guidelines and conclude that there are opportunities to improve their usability. Our work differs in that we use the ISO definition to suggest a theory of refactoring tool usability that we investigate and refine by employing it in the context of developers interacting with refactoring tools. We propose tool improvements that can help usability when making functional changes to software, a common occurrence in the use of these tools.

III. Usability of Refactoring Tools

Existing studies about refactoring tool usability provide insights into a list of detailed factors impacting the use of the tools, but do not provide a framework on which to systematically improve these tools. To bridge this gap, we use a standard definition of usability and apply it to refactoring to provide such a framework, or *theory*. We then use the theory in the remainder of the paper as an "orienting lens [to help guide] what issues are important to examine" [32, p.69] in refactoring tool use.

A. Definition

We base the framework on the ISO 9241-11 [24] standard, which defines usability as *"the extent to which a system, product or service can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use"*.

B. Application

We apply this definition of usability to refactoring tools, considering the user, their goals, and the meaning of effectiveness, efficiency and satisfaction to a user employing one or more of these tools.

The ***user*** of a refactoring tool is a software developer who has access to, and a need for, the tool. Refactoring tools are employed by software developers during all stages of software development, including the creation, evolution and maintenance of software. Developers primarily apply refactoring tools as part of other changes [1]. Both novice and expert developers have easy access to invoke the automation in commonly used development environments during software change tasks, through top-level menu options, keyboard shortcuts and Quick-Assist options directly in editor windows.

The ***goal*** of a developer who applies a refactoring tool is to change the code according to their intent. Several studies find that refactoring most often occurs with the intent of preparing or completing functional changes [4], [5]. This use may seem surprising: why apply a refactoring if the goal is to alter, and not preserve, behavior? Developers choose to use the tools as part of functional changes to avoid making similar, error-prone code changes in many locations [33, §2.2], [3] or to enable reuse of existing code [4]. Thus, the goal is to perform the functional change, and the tool helps achieve it without degrading code quality or introducing errors.

The quality of ***effectiveness*** indicates that a developer can successfully use a refactoring tool to achieve a desired code change. For a developer, effectiveness is a result of their ability to locate and apply desired refactoring operations. This differs from the tool perspective where effectiveness is achieved by ensuring that the tool supports the specifications of refactoring operations.

The quality of ***efficiency*** relates to the overall time or effort that is required to achieve the intended change. Efficiency is impacted by the speed with which the developer can operate the tool and integrate its result into their workflow and the speed with which the tool process files (see e.g. [21]). For example, if the developer has difficulty finding [34] or invoking [12] the correct refactoring operation, it may be more efficient to use a manual process or a simpler tool, such as `grep` [35]. Efficiency can also be impacted by the time that the developer may need to spend locating, verifying or reverting code changes after applying a refactoring tool.

From the tool perspective, efficiency is addressed as the speed required to analyze and change code.

The quality of ***satisfaction*** indicates that the developer experiences using the tool as positive, such as the tool adding value and acting reasonably. Satisfaction is naturally impacted by a lack of effectiveness and efficiency: if error messages produced by the tool are not understandable to the developer or the tool is slow, the developer is unlikely to be satisfied.

C. Theory

This application of the ISO definition of usability to the context of use for refactoring tools leads to the following proposed theory:

> 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 apply for a desired intent (*effective*), reasonable in terms of time and effort required to use the tool (*efficient*), and to add value to the development process (*satisfying*).

IV. Study

To investigate how software developers use refactoring operations as part of their work on software change tasks, we analyze transcripts from a study in which 17 participants, all who had at least a year of professional development experience, attempted three change tasks amenable to the use of refactoring operations. The study was conducted by the authors of this paper. A description of the software change tasks as well as all analysis artifacts are provided in a data package [36].

We present a brief summary of the study here to enable an understanding of the data that formed the basis of this analysis.

A. Study overview

A study session involved a participant sequentially working on three Java-based software change tasks on a non-trivial software system modeled after the Apache Commons-Lang project [37]. The tasks were formulated as software change tasks and did not include refactorings in the instructions. Each participant was asked to think-aloud [25] as they worked. The first author of this paper acted as an observer and interviewer. Screen and audio were recorded with participants' consent.

After each task, a participant was asked three questions: 1) which source code changes were made to solve the task, 2) are there any tools that could have automated the changes, and 3) if any changes may not be correct. At the end of the session, a longer, semi-structured interview was conducted about the participant's experiences with refactoring tools both during the study and previously in their work. The interviewer guided the interview questions by observations made during the tasks, such as the participant's choice to use or disuse refactoring tools, or problems they encountered.[1]

During all interviews, the interviewer used two usability factors—trust and predictability, described in Section II—as sensitizing concepts: if participants spoke about topics related to predictability or trust, the interviewer inquired further, even during tasks.

B. Participants

Participants were recruited in a large North American city. 19 participants were recruited with the requirement of professional Java (or equivalent) programming experience (between 1 and over 20 years, median and average 10 years) and awareness of refactorings. Participants responded to a pre-survey to determine eligibility and give consent. Two participants (P_1 and P_{17}) were excluded from analysis and results after conducting the experiments: one due to technical problems and one due to inability to make progress in the simplest task. From here and out, we refer to "participants" as the remaining 17 participants.

Of the 17 participants, 4 participants were grad students at the time; the remaining 13 were employed by technology companies. 15 participants presented as male and 2 as female. When asked about which IDEs they typically used, 13 participants gave answers that included IntelliJ, 3 participants gave answers that included Eclipse and not IntelliJ, and 1 participant reported only using Sublime. Participants solved tasks in IntelliJ CE 2017 on a provided Macbook Air. IntelliJ provides around 40 refactorings [38]. The task and interview took up to 2 hours per participant.

C. Tasks

The study focused on software change tasks that modeled commits involving refactorings in open-source Java projects. These tasks were mined with a state-of-the-art refactoring detection tool [39] and from the dataset collected in [4].

We iteratively piloted commits we located. Early pilots were conducted in the commit's original projects and showed that participants had difficulties when the code base or domains was unfamiliar. Thus, we chose to map all tasks onto a single codebase that we created, a scaled down version of the Apache Commons-Lang project. This project was one of the mined targets and was chosen for the experiment due to its cohesive code style, its understandable domain (the Java core types), and the relative ease with which a self-contained version could be created. The resulting codebase is of realistic size at 78k lines of Java (v1.8) code and 1335 JUnit tests. The system is built using Maven [40] and version-controlled through git [41]. Table I describes the three tasks used in the study.

[1]To avoid bias, the interviewer avoided introducing these concepts or concrete refactoring operations until the participant did so; these concepts were also introduced at the very end of the interview if they had not already been discussed. Once participants used a particular term for a refactoring or a concept, the interviewer took care to use the same term even if it may not be accurate, and prompted participants to elaborate on what the term meant to them.

TABLE I
TASKS IN STUDY

Task-1	Requires reorganizing test methods by creating a new test class and moving 12 test methods located in two different files into the new class. Relevant refactorings: `move-method`, `extract-class`. Replicates part of an Apache Commons-Lang commit [42].
Task-2	Requires removing two methods, each with a single method that depends on them, and their test methods. Relevant refactorings: `inline-method`. Replicates an Apache Commons-Lang commit [43] that is discussed in this pull-request [44].
Task-3	Requires removing a parameter from eight methods in a single class; each method is overloaded with a wrapper method without the parameter. Relevant refactorings: `inline-method` or `change-signature`. Replicates part of a commit to Quasar [45] that was collected and analyzed by Silva et al. [4].

TABLE II
SUMMARY OF REFACTORING TOOLS INVOKED DURING THE STUDY.

Refactoring	*Code Element Type*	*Count*
Change	Method Signature, Class Signature	36
Inline	Constant, Method	25
Move	Instance Method, Static Method, Class	19
Safe Delete	Method, Parameter	12
Extract	Constant, Superclass	7
Rename	Class	1

D. Data

The study comprised approximately 32 hours of study sessions and contains 100 invocations of refactoring tools by 15 (88%) participants. Five participants (29%) did not use refactoring tools to solve the tasks despite awareness of refactoring tools. Three out of these five participants tried (invoked) tools during interviews after being prompted, but did not use them to solve the tasks. Table II describes which refactoring tools the participants invoked during the study. These tools form the basis of participant experiences during the study. All participants also recalled and reflected on their experiences with refactoring tools they typically use; we consider participants' comments related to tool invocations and their recall of previous experience equally in this paper.

Each study session was screen-captured with audio. As seen in Figure 1, we create a transcript for each session from the recorded screen capture and audio captured information (labelled Data Gathering in Figure 1). To ease analysis, the transcripts are marked with refactoring tool invocations.In total, the 17 transcripts comprise 134,947 words. We use these transcripts as the main data source for analysis.

V. ANALYSIS

We use our theory of refactoring tool usability (Section III) and concepts from related work (Section II) to code the transcripts resulting from the study. We then analyze the resulting codes and associated comments of participants to draw out observations that can explore our theory and guide future refactoring tool designers. All analysis artifacts we use are available to other researchers [36].

TABLE III
CODEBOOK FOR ANALYZING TRANSCRIPTS

Effective	User experiences the tool as successfully performing a desired result.
Efficient	User experiences the tool as being fast or increasing productivity.
Satisfaction	User experiences having their needs or wants met by the use of the tool.
Trust	User experiences the tool as trustworthy, safe or reliable.
Predictable	User understands up-front what will happen by using the tool or indicates a lack of surprise when using the tool.

A. Coding of Transcripts

As our goal in analysis is to refine our theory, we created a codebook from the three usability factors described in the theory—*effectiveness*, *efficiency* and *satisfaction*—and two dominant factors from earlier studies—*trust* and *predictability*. We added these last two factors because of their prevalence in earlier work and because they pertain to the developer's overall experience with refactoring tools. In contrast, other factors noted in earlier work, such as *configuration*, focus primarily on the mechanism or event of refactoring. This approach is consistent with using a theory early in a qualitative analysis [32]. Table III summarizes the codes used in analysis.

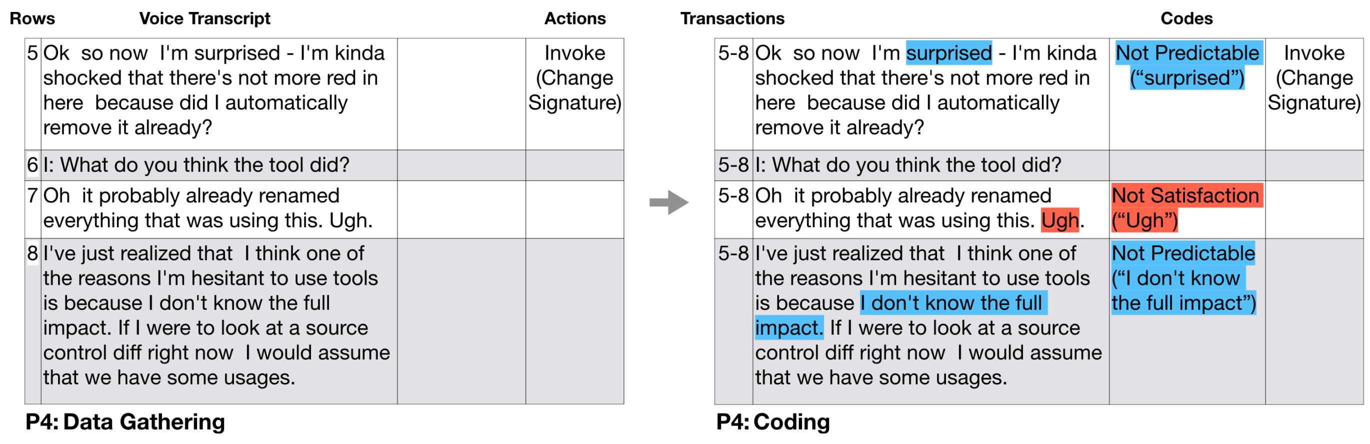
One author of the paper, who also conducted the study sessions, annotated each transcript with the codes from the codebook. A comment made by a participant in a transcript could be assigned more than one code. Each code was recorded either as positive statement or as a negative statement. A second author of the paper reviewed the coding and where disagreement on assigned codes occurred, the two authors discussed the difference until agreement was reached. We used this approach to improve the likelihood of applying the codebook definitions consistently. This approach for reaching agreement has been used elsewhere (e.g., [46]).

As an example, the right-hand side of Figure 1 shows comments with associated codes for a portion of participant #4's (P4's) transcript. In this case, the `change-signature` refactoring tool did not perform as P4 expected ("Not Satisfaction"), changed more than P4 expected ("Not Predictable") and P4 had to resort to using other tools to investigate what the refactoring tool had changed. expressed to be ("Not Efficient").

The coded transcripts are available for others to inspect [36].

B. Analysis of Codes

A naive analysis of the coded transcripts could lead to an over-approximation of some codes as the ways in which participants expressed usability factors in comments varied. Some participants were sparse in their descriptions, whereas others were more verbose. To help normalize across descriptions, we manually grouped sequences of coded comments in the transcript if they represented one thought of a participant. For ease of reference, we refer to a grouped set of comments as a *transaction*. For example, all comments in Figure 1 (rows



P4: Data Gathering

Rows	Voice Transcript		Actions
5	Ok so now I'm surprised - I'm kinda shocked that there's not more red in here because did I automatically remove it already?		Invoke (Change Signature)
6	I: What do you think the tool did?		
7	Oh it probably already renamed everything that was using this. Ugh.		
8	I've just realized that I think one of the reasons I'm hesitant to use tools is because I don't know the full impact. If I were to look at a source control diff right now I would assume that we have some usages.		

P4: Coding

Transactions		Codes	
5-8	Ok so now I'm surprised - I'm kinda shocked that there's not more red in here because did I automatically remove it already?	Not Predictable ("surprised")	Invoke (Change Signature)
5-8	I: What do you think the tool did?		
5-8	Oh it probably already renamed everything that was using this. Ugh.	Not Satisfaction ("Ugh")	
5-8	I've just realized that I think one of the reasons I'm hesitant to use tools is because I don't know the full impact. If I were to look at a source control diff right now I would assume that we have some usages.	Not Predictable ("I don't know the full impact")	

Fig. 1. Excerpt of transcript from participant 4 as seen in the data gathering phase (on the left) and coding step (on the right).

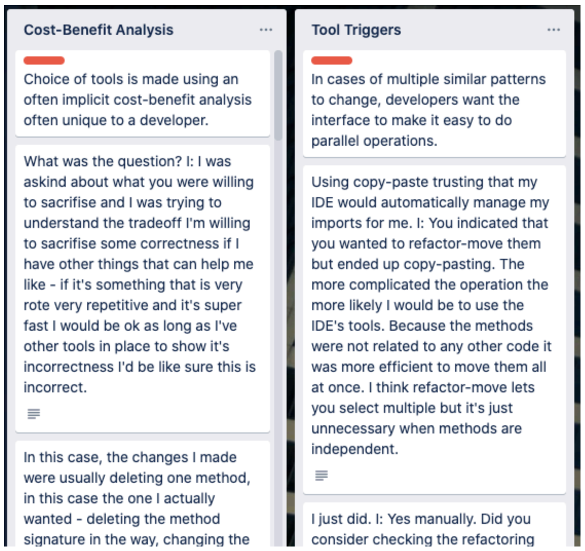
5-8 of the transcript) group into the same transaction as they refer to *thoughts about* the same tool invocation.

To support the investigation of the predominance of usability factors across the experience of participants, we use association rule mining [47], which reports itemsets—co-occuring sets of codes—above a pre-defined support level, where support is the number of transactions containing the subset of codes. An itemset of size one indicates occurrences of one particular code; an itemset of size two indicates two co-occurring codes, and so on. We apply the *apriori* algorithm [48] as found in the python `mlxtend` library [49] to find association rules in the transactions from the coded transcripts. When applying association rule mining, we consider the codes both with sentiments attached (e.g., "Satisfaction" and "Not Satisfaction") and not attached. We also count each code only once per transaction even if there are several occurrences of the code. For example, the transaction in Figure 1 is coded as {Not Predictable, Not Satisfaction} with sentiment and {Predictable, Satisfaction} without sentiment. The use of association rule mining lets us investigate if one usability factor may indicate the presence of other usability factors, as in whether predictability likely indicates satisfaction.

C. Analysis of Comments

To guide an investigation of the participants' views about refactoring tool usability, we use the comments organized by codes. Our aim is to find themes across the transactions for an itemset. Card sort [26] is an approach to detecting themes in qualitative data where one or more individual organize cards into groups until themes emerge. This is comparable to the thematic analysis employed in [50] whereas due to the richness of our data, we perform one individual card sort for each itemset of size one (i.e., each code) instead of searching for themes across all comments simultaneously.

We create "cards" (using Trello [51]) where each card represents a transaction (grouped set of comments) associated with that itemset; for each transaction, we include on the card comments immediately preceding and immediately succeeding the transaction to provide sufficient context for interpreting



Cost-Benefit Analysis

Choice of tools is made using an often implicit cost-benefit analysis often unique to a developer.

What was the question? I: I was askind about what you were willing to sacrifise and I was trying to understand the tradeoff I'm willing to sacrifise some correctness if I have other things that can help me like - if it's something that is very rote very repetitive and it's super fast I would be ok as long as I've other tools in place to show it's incorrectness I'd be like sure this is incorrect.

In this case, the changes I made were usually deleting one method, in this case the one I actually wanted - deleting the method signature in the way, changing the

Tool Triggers

In cases of multiple similar patterns to change, developers want the interface to make it easy to do parallel operations.

Using copy-paste trusting that my IDE would automatically manage my imports for me. I: You indicated that you wanted to refactor-move them but ended up copy-pasting. The more complicated the operation the more likely I would be to use the IDE's tools. Because the methods were not related to any other code it was more efficient to move them all at once. I think refactor-move lets you select multiple but it's just unnecessary when methods are independent.

I just did. I: Yes manually. Did you consider checking the refactoring

Fig. 2. Snippet of Card Sort for Transactions including Efficiency code. Cards without labels are transaction comments from transcripts; cards with labels (the two topmost cards) are theme-cards.

the information. Figure 2 shows a snippet of the card sort for transactions coded with "Efficiency".

By employing one card sort per itemset, we find themes that emerge from comments pertaining to that usability factor. As it is likely that some themes will be similar across factors, we look for common themes across the card sorts and "collapse" them together by performing a "meta-card sort". In the meta-card sort, each card is a theme from the previous iteration. For example, the theme "Cost-Benefit Analysis" in the "Efficiency" card sort in Figure 2 is a single card in the meta-card sort.

Two authors of the paper perform the card sorts, discussing and resolving differences of opinion.

VI. Results

Our coding of the transcripts resulted in comments being labelled with 284 codes from which we form 143 transactions.

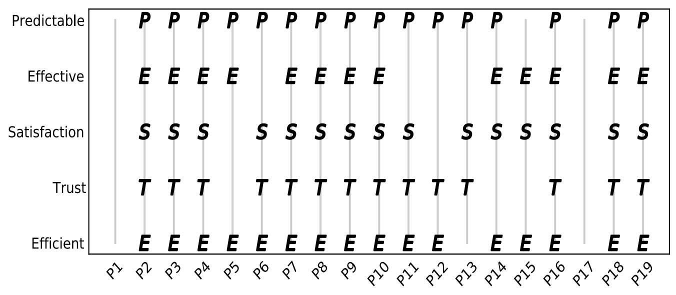


Fig. 3. If the transcript of a participant (x-axis) contains one or more statements labeled with a usability factor (y-axis), then we mark the coordinate (participant, factor) with the first letter in the factor. For example, P_6 has a transaction labeled Predictable and is marked with a P. P_6 did not have a transaction marked Effective, so that coordinate is left blank. This figure shows that factors were distributed across the participants and that frequent factors like Predictable and Efficient were represented in statements from almost all participants.

We report on the prevalence of codes and themes about usability factors based on this data. Figure 3 shows that the usability factors were distributed across participants.

A. *Prevalence of Usability Factors*

Table IV reports the itemsets found by applying the *apriori* association rule mining algorithm to the codes in the transcripts when sentiments associated with the codes are ignored. We used a support of 0.15 when applying the mining algorithm as a means of finding itemsets that appear in at least 15% of the transactions. The resulting itemsets are shown in Table IV.

The top row indicates that in over half of the transactions coded, participants refer to predictability of the refactoring tools. The prevalence of discussion of this usability factor over others indicate that refactoring tool designers may wish to pay special attention to how developers view the predictability of actions of a tool when invoked. In contrast, while participants also referred to trust, they did so the least of all factors. This data starts to bring out differentiations of usability factors to explore relative to earlier research that reported on multiple usability factors uniformly.

Observation #1: Participants commented frequently on the predictability of refactoring tools.

Table IV also includes three itemsets with more than one usability factor. Predictability is present in two out of three itemsets and occurs together with satisfaction (in 18% of transactions) and effectiveness (in 15% of transactions). Interestingly, participants did not often refer to more than one usability factor (e.g., support values are less than 0.2). It is also interesting that participants did not often refer to efficiency or trust together with other factors: all three itemsets of size two involve predictability, satisfaction and effectiveness. We hypothesize that focusing on mechanisms that support these usability factors may help refactoring tool designers deliver tools of more interest and value to software developers.

We also ran the *apriori* algorithm on sentiment attributed

TABLE IV
ITEMSETS WITH NON-SENTIMENT ATTRIBUTED CODES

{Predictable}	0.55
{Effective}	0.38
{Satisfaction}	0.36
{Efficient}	0.29
{Trust}	0.21
{Predictable, Satisfaction}	0.18
{Satisfaction, Effective}	0.16
{Predictable, Effective}	0.15

TABLE V
CARD SORT INFORMATION FOR ITEM SETS OF SIZE ONE (INDIVIDUAL USABILITY FACTORS).

Cardsorts	Cards	Discard	Themes
Predictable	79	27	7
Effective	53	22	6
Satisfaction	51	19	5
Efficient	42	12	5
Trust	31	7	5
Total	256	87	28

codes, such as when a participant's comments were coded as "Not Satisfaction". The top two results had negative sentiments attached: "Not Predictable" with support of 0.31 and "Not Satisfaction" with support of 0.27. We note that participants may be more likely to vocalize negative sentiments. Successful use of tools, such as `rename`, did not lead to many comments; participants were more vocal when encountering frustrating or confusing tools.

B. *Usability Factor Themes*

To gain a deeper understanding of the themes underlying participants' comments, we performed five cardsorts on each code (itemsets of size one in Table IV). Table V shows the number of cards considered in each card sort, the number of cards discarded and not associated with a theme, and the number of themes identified. For example, for the Predictable card sort, 79 cards were considered, of which 27 (34%) were discarded. A card was discarded if the comments on the card did not provide a coherent insight about refactoring tool use. From the cards considered, 7 themes were identified for the Predictability usability factor. The results of the card sorts are available in the data package for this paper [36].

To give a sense of the emergent themes, we describe a theme from the Predictable card sort and from the Efficent card sort.

One of the seven themes from the predictable card sort is *review and convince*. This theme describes 15 cards from 10 participants. This theme summarizes comments made by participants that expressed how participants wanted to check how a refactoring tool changed the code after invocation. Participants wanted to perform this check as a means of convincing themselves that the tool made changes that the participant expected. We saw several participants use other tools, such as git diff, to accomplish such a check. Participant P7 commented:

> *"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."*

Observation #2: Participants rely on ad-hoc solutions to understand code changes after applying a refactoring.

As a second example, one of the five themes from the efficient card sort is *cost-benefit analysis*. This theme summarizes 22 cards from 13 participants. This theme describes that participants evaluated the benefits of using the tool against the cost of using it. For instance, participant P8 tried using `change-signature` during Task-3. During the interview they said

> *"I tried using the refactor tool, change signature. It is all right, it is useful I think when you are changing a lot of places. In some cases, it was not so useful because it was as much work to run that as to go and change those places."*

Observation #3: Participants lacked support for up-front cost-benefit analysis of applying a refactoring tool.

C. Usability Themes

The meta-card sort across the 28 themes resulted in 4 themes, listed in Table VI. Two of these themes capture requests about what refactoring tools need to convey to developers: namely to make the capabilities of the tools more apparent to a developer and to better communicate to a developer what the tool does as, and after, the tool is invoked. The other two themes capture additional interactions that developers desire to have with refactoring tools, namely to guide how a tool operates and to enable a developer to better assess whether to use the tool or not.

VII. Discussion

We consider how the results of our analysis impact the theory of refactoring tool usability introduced earlier in the paper, implications for designers of refactoring tools, the relationship of our insights into usability theories and factors to earlier work, and threats to the study results on which this paper relies.

A. Theory of Refactoring Tool Usability: Revisited

In Section III, we present a theory of refactoring tool usability derived from the ISO 9241-11 definition of usability. We use the insights gained from the study and resultant analysis of participants' comments to refine this theory. In particular, we note that the four themes presented in Table VI cover the comments coded as representing effectiveness, satisfaction and efficiency. We use these themes to revisit and refine a theory of refactoring tool usability proposing the following as a refined theory:

> 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.

B. Implications for Refactoring Tool Designers

Over the years, substantial improvements have been made to refactoring tools to improve their usability. For example, many refactoring tools now include 'preview' windows that provide a glimpse into what parts of the code will be changed by applying the operation. As another example, some refactoring tools present options to a developer: `inline-method` in IntelliJ lets the developer choose between:

- Inline one caller and keep the declaration unchanged.
- Inline all callers and keep the declaration unchanged.
- Inline all callers and delete the declaration.

These tool interfaces start exposing the complexity of refactoring operations and can be seen as inviting a dialogue between the software developer (the user) and the refactoring tool.

The observations we made during the study and themes we drew out about tool usability indicate three major and one minor set of implications for refactoring tool designers to hone the dialogue between developer and tool.

Up-front Impact. The first major implication, based on the 'Developer cost-benefit analysis' and 'Tool communications change' themes (Table VI), is that refactoring tools need to communicate much more information about the potential impact of the refactoring at the start of the refactoring process. This information is necessary for two reasons: 1) users must be able to recognize the *impact* that a change will have on their code to understand if they want to apply it, and 2) users must be able to evaluate the *cost and benefit* of using the automated tool to apply the change rather than using a manual approach. Both of these needs relate to deciding whether to apply the refactoring.

At first glance, one might think this is precisely the reason for 'preview' windows. We found through the study that the information provided by 'preview' windows was insufficient for both purposes. Even developers who engaged with the 'preview' window were surprised at the impact of the refactoring after they were applied, or ended up applying refactorings that introduced unwanted changes, which subsequently needed to be reverted or fixed. Using our dialogue metaphor, developers wanted a richer start to the conversation with which they engage a refactoring tool.

Guided Change. A second major implication, based on the 'Developer guides tool' theme (Table VI) is that once a

TABLE VI
THE USABILITY THEMES THAT EMERGE IN THE META-CARD SORT OF THE 28 INITIAL THEMES.

Theme	Description	Predictable	Effective	Satisfaction	Efficient	Trust
Tool communicates capabilities	A tool's user interface must communicate clearly and directly to a developer in terms familiar to a developer. The tool should guide a developer in its use, including providing intelligible error messages.	✓		✓		
Tool communicates change	A tool should make the changes it will make to code clear before the tool is executed. It should be possible for a developer to inspect the changes that the tool has made.	✓	✓	✓		✓
Developer guides tool	Developers wish to guide how a tool executes in several ways: applying an operation to many elements easily, excluding the application to some locations of code, and altering how particular code is changed.	✓	✓	✓	✓	✓
Developer cost-benefit analysis	Developers assess the cost of applying a tool before they invoke the tool. Developers want to assess whether it is better to proceed with a tool or manually at the start of considering using a tool. The tool should help them with this assessment.	✓	✓		✓	

developer decides to proceed with a refactoring, they want control of where and how a refactoring is applied. Refactoring tool designers should consider how to enable a developer to step through code changes associated with a refactoring, similar to how a step function in a debugging tool works. In each step, the developer could inspect the code location, inspect the change proposed by the refactoring tool, and perform potentially location-specific code alterations.

This approach would solve several usability problems we identified through the study: it would let the developer perform *additional changes* on each location; it would guide the developer through the code so that they *gather knowledge* about it; it would allow the developer to *validate* each code change as it was applied; it would help the developer understand the tool and increase trust.

Of course, enabling developers to skip or change the refactoring tool's intended change at each step also means that the final code may not have preserved behaviour, contradicting the safety guarantee of the refactoring tool!

Perhaps surprising to some, this lack of safety is already possible depending on choices developers make in 'preview' windows and options provided by existing refactoring tools. For example, invoking `move-method`on a JUnit test method in IntelliJ, causes the refactoring tool to reject the invocation, and to propose to make the method static and move it. If the developer agrees, the method becomes static and the developer will cease to be able to execute it as a test method. The result is a behaviour change despite using a refactoring tool. P19 encountered this situation in Task-1 but failed to recognize that he agreed to an unsafe change. We see opportunities for refactoring tool designers to walk developers through changes and explaining impacts of skipping or performing code changes suggested by a refactoring tool.

There is an interesting design space to explore between safety guarantees that can lead developers to disuse tools and unsafe code change support that developers might choose to use. By addressing this implication, tool designers can give the user more control of how the conversation unfolds.

Communicating Change. A third major implication, based on the 'Communicating Change' theme (Table VI), is the need for developers to understand what a refactoring tool did to their code. Numerous times in our study, we saw developers resort to ad-hoc solutions and additional tools (e.g., git) to understand what a refactoring tool had changed in their code. Participants were not always satisfied with this solution. P19, for instance, said:

> *"Sometimes inspecting changes after the fact can be hard. There is going to be code that is unrelated to your refactoring, since normally refactoring tasks is done as part of other tasks. When you're looking at it after the fact you may also not be able to undo it if something doesn't look right ..."*

Refactoring tool designers should consider how to provide summaries of their impact on the code. Perhaps this could be as simple as what is provided by a git diff operation. Or, perhaps a user could replay changes made by the refactoring tool to understand why changes are being made. This support can be considered a summary provided at the end of a conversation between the user and the refactoring tools.

Initiating Use. A more minor implication than the other three is how to start a conversation between the user and the appropriate refactoring tools. We observed developers struggle to understand what a refactoring operation might be, and to then invoke the desired refactoring operation. For example, in Task-1, participant P2 made many attempts to invoke `move-method`, but failed to do so because their model of how they should invoke the tool differed from the tool's affordances, and the mismatch was not clear. Specifically, the participant made a text-selection across multiple methods as they wanted to invoke `move-method` on all of them and thought that the tool would recognize their intent; the tool recognized that the menu was invoked on the class body and invoked `move-class` instead, which resulted in an incomprehensible error message. The participant finally realized that they were invoking the wrong type of move and moved on to `extract`.

> *"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. 'Cause that implies you just got a small section of code."*

Participant P18 had a similar experience that lead to them introducing a bug. These participants had the following problem: the tool failed to communicate which actions mattered (the location of the right-click) and what didn't (the selected area). Therefore, they were unable to locate and invoke the functionality they wanted.

C. Relationship to Earlier Work

As described in Section II, multiple studies have made observations about factors that contribute to use and disuse of refactoring tools, such as developers lack trust in tools, the tools are unpredictable, developers lack awareness and familiarity with tools, the tools disrupt their workflow and do not support operations they need. Despite this list of observations, previous work provides few actionable items. In particular, there is little known about how to make tradeoffs between different factors, such as supporting more operations (which can be improved by implementing more refactorings) and predictability (which can be improved by implementing fewer and simpler refactorings). By forming a theory about usability and investigating developer experiences through that theory, this paper presents themes capturing developer desires and expectations from refactoring tools that can be used to design the next generation of refactoring tools.

The four usability themes we derive from the study we conducted (Table VI) encapsulate a number of steps in a refactoring process. Some of these steps are not included in the process described by Mens and Tourwe [27], such as having the tool better communicate the change and having more ability for a developer to guide how the tool proceeds with changes. The latter step is present in the usage pattern presented in Murphy-Hill's model of refactoring tool use [28], where the developer reacts to the output from the tool. The dialogue metaphor that captures many of these steps and that we discuss above aligns with findings by Vakilian et al. [2] and claims by Brant [22] that developers accept unsafe tools as long as the tool provides value. In these ways, our findings confirm previous work while also extending with new observations and providing a framework in which to consider multiple aspects of usability together.

D. Threats to Validity

Our investigation of usability factors for refactoring tools is dependent upon the context in which participants experienced refactoring tools. To mimic reality in our study, we modeled the three change tasks based on scenarios mined from open source systems. We created a non-trivial sized system (78k lines of code) to enable a target system understandable to a broad set of participants in the study. Similar to reports of others, we saw evidence of refactoring tool disuse as five participants did not use any available refactoring tools despite showing awareness of refactoring operations. Furthermore, 10 out of 17 participants proposed solutions to the tasks that avoided using refactoring tools. Finally, six out of 17 participants shared (without prompt) either that the tasks were realistic or that they had encountered such tasks before during their work. Thus, although there are threats to the generalizability of our results, a number of mitigation steps were taken.

The analysis we conducted is predicated on the coding of transcripts and grouping of coded comments into transactions. We mitigated this threat by having two authors review, assess and agree upon the codes and groupings. We took a similar approach with the card sorts used to identify emergent themes. When using techniques such as card sort, bias of the researchers is a threat. As a further mitigation, we make all of this data available to allow other researchers to interrogate.

During the interviews, we observed that successful usages of refactoring tools were less commented upon by participants. In contrast, when the tool did not work or the operations were complex, participants were more vocal. This may introduce bias in the results.

VIII. Summary

Despite the prevalence and availability of refactoring tools across many development environments, software developers do not use these tools.

In this paper, we have sought to investigate how to improve the usability of refactoring tools using the lens of a theory we derived from an ISO standards definition of usability. Using this theory, we study the experiences of 17 developers who we asked, in a lab study, to work on three change tasks designed to be amendable to the use of refactoring tools. Our analysis resulted in four emerging themes about the usability of refactoring tools, which we used to refine our theory. These themes provide insights into what developers desire and need from refactoring tools. In particular, these themes suggest that refactoring tool designers need to consider how refactoring tools provide information to a developer and how developers can be given more control about how refactoring tools operate.

This paper also provides insights for software engineering researchers. First, the paper provides a theory of refactoring tool usability for other researchers to build upon. Second, the paper introduces a systematic means of considering usability factors of software development tools from a user's perspective instead of a tool's perspective; this perspective change can lead to a questioning of implicit assumptions of what tool designers may believe (i.e., refactoring tools must be safe) and what developers want (i.e., the reduced invocation-cost of an unsafe change). We hope that this might enable the building of tools (refactoring or others) that are used more often and that developers find effective, efficient and satisfying.

References

[1] E. Murphy-Hill, C. Parnin, and A. P. Black, "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, pp. 287–297. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2009.5070529

[2] M. Vakilian, N. Chen, S. Negara, B. A. Rajkumar, B. P. Bailey, and R. E. Johnson, "Use, disuse, and misuse of automated refactorings," in *2012 34th International Conference on Software Engineering (ICSE)*, June 2012, pp. 233–243.

[3] M. Fowler, *Refactoring: improving the design of existing code*. Addison-Wesley, 1999.

[4] D. Silva, N. Tsantalis, and M. T. Valente, "Why we refactor? confessions of github contributors," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: ACM, 2016, pp. 858–870. [Online]. Available: http://doi.acm.org/10.1145/2950290.2950305

[5] M. Paixão, A. Uchôa, A. C. Bibiano, D. Oliveira, A. Garcia, J. Krinke, and E. Arvonio, "Behind the intents: An in-depth empirical study on software refactoring in modern code review," *17th MSR*, 2020.

[6] W. F. Opdyke, "Refactoring object-oriented frameworks," Ph.D. dissertation, 1992.

[7] "Eclipse ide," https://www.eclipse.org, [Online: accessed 25-October-2020].

[8] "Intellij ide," https://www.jetbrains.com, [Online: accessed 25-October-2020].

[9] C. Abid, V. Alizadeh, M. Kessentini, T. do Nascimento Ferreira, and D. Dig, "30 years of software refactoring research:a systematic literature review," 2020.

[10] M. Kim, T. Zimmermann, and N. Nagappan, "A field study of refactoring challenges and benefits," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE '12. New York, NY, USA: ACM, 2012, pp. 50:1–50:11. [Online]. Available: http://doi.acm.org/10.1145/2393596.2393655

[11] S. Negara, N. Chen, M. Vakilian, R. E. Johnson, and D. Dig, "A comparative study of manual and automated refactorings," in *European Conference on Object-Oriented Programming*. Springer, 2013, pp. 552–576.

[12] E. Murphy-Hill and A. Black, "Breaking the barriers to successful refactoring," in *2008 ACM/IEEE 30th International Conference on Software Engineering*, May 2008, pp. 421–430.

[13] K. Narasimhan and C. Reichenbach, "Copy and paste redeemed (t)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 630–640.

[14] Y. Y. Lee, N. Chen, and R. E. Johnson, "Drag-and-drop refactoring: Intuitive and efficient program transformation," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 23–32. [Online]. Available: http://dl.acm.org/citation.cfm?id=2486788.2486792

[15] E. Murphy-Hill and A. P. Black, "High velocity refactorings in eclipse," in *Proceedings of the 2007 OOPSLA Workshop on Eclipse Technology eXchange*, ser. eclipse '07. New York, NY, USA: ACM, 2007, pp. 1–5. [Online]. Available: http://doi.acm.org/10.1145/1328279.1328280

[16] C. Reichenbach, D. Coughlin, and A. Diwan, "Program metamorphosis," in *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, ser. Genoa. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 394–418. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-03013-0_18

[17] M. Vakilian, N. Chen, R. Z. Moghaddam, S. Negara, and R. E. Johnson, "A compositional paradigm of automating refactorings," in *European Conference on Object-Oriented Programming*. Springer, 2013, pp. 527–551.

[18] S. R. Foster, W. G. Griswold, and S. Lerner, "Witchdoctor: Ide support for real-time auto-completion of refactorings," in *2012 34th International Conference on Software Engineering (ICSE)*, June 2012, pp. 222–232.

[19] X. Ge, Q. L. DuBose, and E. Murphy-Hill, "Reconciling manual and automatic refactoring," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 211–221. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337223.2337249

[20] X. Ge and E. Murphy-Hill, "Manual refactoring changes with automated refactoring validation," in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 1095–1105.

[21] J. Kim, D. Batory, D. Dig, and M. Azanza, "Improving refactoring speed by 10x," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 2016, pp. 1145–1156.

[22] J. Brant and F. Steimann, "Refactoring tools are trustworthy enough and trust must be earned," *IEEE Software*, vol. 32, no. 6, pp. 80–83, Nov 2015.

[23] J. Oliveira, R. Gheyi, M. Mongiovi, G. Soares, M. Ribeiro, and A. Garcia, "Revisiting the refactoring mechanics," *Information and Software Technology*, vol. 110, pp. 136 – 138, 2019. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950584919300461

[24] Ergonomics of human–system interactionpart 11: Usability: Definitions and concepts iso 9241-11: 2018 (en). (Accessed September 10th, 2020). [Online]. Available: https://www.iso.org/obp/ui/#iso:std:iso:9241:-11:en

[25] M. Van Someren, Y. Barnard, and J. Sandberg, "The think aloud method: a practical approach to modelling cognitive," *London: AcademicPress*, 1994.

[26] J. Corbin and A. Strauss, *Basics of qualitative research: Techniques and procedures for developing grounded theory*. Sage publications, 2014.

[27] T. Mens and T. Tourwe, "A survey of software refactoring," *IEEE Transactions on Software Engineering*, vol. 30, no. 2, pp. 126–139, Feb 2004.

[28] E. Murphy-Hill, "A model of refactoring tool use," *Proc. Wkshp. Refactoring Tools*, 2009. [Online]. Available: https://people.engr.ncsu.edu/ermurph3/papers/wrt09.pdf

[29] D. Campbell and M. Miller, "Designing refactoring tools for developers," in *Proceedings of the 2nd Workshop on Refactoring Tools*, ser. WRT '08. New York, NY, USA: ACM, 2008, pp. 9:1–9:2. [Online]. Available: http://doi.acm.org/10.1145/1636642.1636651

[30] G. A. Bowen, "Sensitizing concepts," *SAGE Research Methods Foundations. SAGE Publications. doi*, vol. 10, no. 9781526421036788357, 2019.

[31] E. Mealy, D. Carrington, P. Strooper, and P. Wyeth, "Improving usability of software refactoring tools," in *2007 Australian Software Engineering Conference (ASWEC'07)*. IEEE, 2007, pp. 307–318.

[32] J. W. Cresswell and J. D. Creswell, *Research Design: Qualitative, Quantitative and Mixed Methods Approaches*, 5th ed. SAGE, 2018.

[33] E. Murphy-Hill, "Programmer friendly refactoring tools," Ph.D. dissertation, 2009.

[34] E. Murphy-Hill, M. Ayazifar, and A. P. Black, "Restructuring software with gestures," in *2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, Sep. 2011, pp. 165–172.

[35] "grep," https://www.gnu.org/software/grep/, [Online: accessed 12-October-2020].

[36] A. M. Eilertsen and G. C. Murphy, "Data Package," https://github.com/annaei/Replication-Data-for-The-Usability-or-Not-of-Refactoring-Tools/, [Online: accessed 08-January-2021].

[37] "Apache commons-lang," commons.apache.org/proper/commons-lang, [Online: accessed 25-August-2020].

[38] "Intellij refactoring," https://www.jetbrains.com/help/idea/refactoring-source-code.html, [Online: accessed 01-January-2021].

[39] N. Tsantalis, M. Mansouri, L. M. Eshkevari, D. Mazinanian, and D. Dig, "Accurate and efficient refactoring detection in commit history," in *Proceedings of the 40th International Conference on Software Engineering*. ACM, 2018, pp. 483–494.

[40] "Maven," https://maven.apache.org/, [Online: accessed 28-August-2020].

[41] "git," https://git-scm.com/, [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," https://github.com/apache/commons-lang/commit/3ce7f9eecfacbf3de716a8338ad4929371a66ca2, [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] "Quasar Commit 56d4b99," https://github.com/puniverse/quasar/commit/56d4b999e8be70be237049708f019c278c356e71, [Online: accessed 24-July-2020].

[46] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 672–681.

[47] R. Agrawal, T. Imieliński, and A. Swami, "Mining association rules between sets of items in large databases," *SIGMOD Rec.*, vol. 22, no. 2, p. 207216, Jun. 1993.

[48] R. Agrawal and R. Srikant, "Fast algorithms for mining association rules in large databases," in *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases*, 1994, pp. 487–499.

[49] "mlxtend library," https://rasbt.github.io/mlxtend, [Online: accessed 25-October-2020].

[50] J. Sillito, G. C. Murphy, and K. De Volder, "Questions programmers ask during software evolution tasks," in *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*. ACM, 2006, pp. 23–34.

[51] "Trello," http://trello.com, [Online: accessed 25-October-2020].