A Study on Developer Perception of Transformation Languages for Refactoring

Christian D. Newman
Software Engineering
Rochester Institute of Tech
Rochester, NY 14623 USA
cnewman@se.rit.edu

Mohamed Wiem Mkaouer
Software Engineering
Rochester Institute of Tech
Rochester, NY 14623 USA
mwmvse@rit.edu

Michael L. Collard
Computer Science
The University of Akron
Akron, OH USA
collard@uakron.edu

Jonathan I. Maletic
Computer Science
Kent State University
Kent, OH 44240 USA
jmaletic@kent.edu

ABSTRACT

Although there is much research advancing state-of-art of program transformation tools, their application in industry source code change problems has not yet been gauged. In this context, the purpose of this paper is to better understand developer familiarity and comfort with these languages by conducting a survey. It poses, and answers, four research questions to understand how frequently source code transformation languages are applied to refactoring tasks, how well-known these languages are in industry, what developers think are obstacles to adoption, and what developer refactoring habits tell us about their current use, or underuse, of transformation languages. The results show that while source code transformation languages can fill a needed niche in refactoring, research must motivate their application. We provide explanations and insights based on data, aimed at the program transformation and refactoring communities, with a goal to motivate future research and ultimately improve industry adoption of transformation languages for refactoring tasks.

CCS CONCEPTS

• **General and reference→Surveys and overviews**

KEYWORDS

Software Maintenance, Software Evolution, Source Code Manipulation, Refactoring

ACM Reference format:

Christian D. Newman, Mohamed Wiem Mkaouer, Michael L. Collard, and Jonathan I. Maletic. 2018. A Study on Developer Perception of Transformation Languages for Refactoring. In Proceedings of the 2nd International Workshop on Refactoring (IwoR '18), September 4, 2018, Montpellier, France. ACM, New York, NY, USA, *8 pages. https://doi.org/10.1145/3242163.3242170*

1 INTRODUCTION

As software ages, it must be updated to conform with new requirements, changing environments (e.g., updated operating system, new API), and other shifting development needs. These all fall under the umbrella of software maintenance, which is the costliest phase of the software development lifecycle. One common activity to maintain software is refactoring [1]. Refactoring is a change applied to the structure of source code which 1) does not change the observable behavior and 2) improves code characteristics such as comprehensibility and complexity.

Developers may apply refactorings manually or automatically; performing and testing changes to the code by hand or using a tool to execute changes in a (semi) automated, systematic fashion. In the situation where the change to be applied is to a large body of code, manually refactoring is expensive and error prone [2, 3]. For this reason, it seems reasonable to assume automated tools for refactoring would be popular in development, or at least significantly more prevalent than manual application when a tool is available to automate the change. However, numerous studies have shown that developers do perform manual refactorings more often [3-5]. That is, standard refactoring tools are widely underused considering the theoretical and proven benefits (i.e., faster, highly consistent changes, no missed changes).

The research presented in [3-5] focuses broadly on refactoring tools without differentiating between types of them. In this paper, we discuss two different types of refactoring tools. The first type represents standard refactoring tools; these are tools that are common in IDEs such as Visual Studio, Eclipse, etc. These tools define a finite set of refactorings that are apllied by having the developer fill out a set of fields. Some IDE's additionally allow the user to leverage an API and a general-purpose language (e.g., Python) to help define refactorings. An alternative type of refactoring approach involves the use of transformation languages; languages such as RASCAL [6], TXL [7], and srcML/srcTL[8]. A transformation language differs from a standard refactoring tools as it uses a Domain Specific Language (DSL) as the primary way of describing and applying a refactoring.

Transformation languages have been involved as a source of automation for refactoring [2, 9], so their applicability to refactoring problems is known in research and in some parts of the industry. The core advantages to using transformation languages are: 1) the flexibility to define custom refactorings using transformation languages; and 2) a DSL that is specialized to make the definition of transformations clean and easy to comprehend/re-use. That is, a specialized DSL for transformation can be used to define custom refactorings, which is more flexible than many standard refactoring tools. Moreover, since the DSL is specialized for the domain of transformations, it should be easier to comprehend than writing an equivalent custom refactoring using a general-purpose language.

The need for the flexibility of these languages is supported by interest in techniques that automatically generate transformations, all of which emphasize learning a change, such as refactoring, and automatically applying it in an unseen context. Additionally, in previous research, developers have reported that automatically

IWOR'18, September , 2018, Montpellier, France

ACM ISBN 978-1-4503-5974-0/18/09...$15.00
https://doi.org/10.1145/3242163.3242170

applied refactorings tend to 1) be small and part of a larger, manually applied refactorings task and 2) that one challenge associated with refactoring is the lack of tool support for defining new refactoring types [10]. This, along with research on generating transformations, implies that standard refactoring tools do not fully encompass the breadth or complexity of the refactorings developers need to handle in practice.

In this paper, we investigate the perceived lack of transformation languages in use by developers and compare it with the usage of refactorings tools. Specifically, we surveyed 50 developers; asking, for example, which transformation languages and standard refactoring tools they are familiar with, which they have used, and what types of refactorings they commonly apply in their day-to-day operations. To the best of our knowledge, this is the first survey that aims to understand the use of transformation languages for refactoring in an industrial setting. We answer the following research questions:

1. Are refactoring tools more well-known than source code transformation languages?
2. How often do developers use transformation languages versus standard refactoring tools in their development activities?
3. What do developers perceive as obstacles to the adoption of transformation languages and refactoring tools?
4. What standard refactoring practices our participants do perform?

The rest of this paper is structured as follows. Section 2 motivates the problem, Section 3 presents our Research Methodology, Section 4 discusses the results of the survey, Section 5 discusses the study limitations, and Section 6 concludes.

2 MOTIVATION

The problem with manual source code changes for refactorings, and in particular applications of refactorings to large-scale systems, is that they are time-consuming and error-prone. Developers must search (possibly millions of lines of code) for the appropriate segments of code that need to be refactored, develop a solution for each individual situation (combining and reusing solutions when possible), apply the change, perform regression tests to guarantee consistent behavior with the previous version, and then integrate the code with the main branch once everything has been finalized.

Reducing the manual maintenance effort while refactoring has been the focus of a recent study [2]. We examine this work more closely as a documented example of the advantages of using an automated approach in an industry setting. In that paper, the authors explored several maintenance problems faced by ABB Inc. and designed refactorings in a transformation language (XSLT) to resolve a set of problems. One of the problems they explored occurs due to updates to the C++ standard. Several years ago, a change to the C++ standard dictated that C++'s operator *new* should throw an exception instead of its previous behavior; returning 0 or null. This caused the need for adaptive maintenance; maintenance whose goal is to update the code base due to a change in the environment. Initially, ABB solved this problem manually. They assigned developers to the task of changing all calls to operator new such that they were wrapped in a try-catch block instead of checking their return value for null.

The original change encompassed around 1.7K individual calls to operator *new* and took one developer approximately a month to fully complete (i.e., applying each change, regression testing, code review, re-integration). However, the developer missed approximately 300 cases of the call to operator *new* in their original effort. These 300 cases were eventually corrected through future changes to the code base.

We highlight this work as a clear example that the manual effort is both time-consuming and difficult to carry out. Collard et al. took the original code (i.e., before the fixes were applied) and used a program transformation constructed by combining the srcML format [11] with XSLT; a transformation language for XML. Their transformation approach fixed all 1.7k calls to operator *new* in around 11 minutes and did not miss any cases. As such, this is a clear example of the advantages of the automated process. Not only did it apply changes faster and more consistently, but also the transformation script is reusable.

TABLE 1. PARTICIPANT PROFESSIONAL DEVELOPMENT EXPERIENCE IN YEARS

Years of Experience	%	Count
0-2	8.82%	7
3-5	26.47%	14
6-10	20.59%	8
11+	44.12%	21
Total	100.00%	50

TABLE 2. PARTICIPANT LEVEL OF EDUCATION

Level of Education	%	Count
Less than high school	0.00%	0
High school graduate	2.94%	1
Some college	8.82%	4
2-year degree	2.94%	2
4-year degree	38.24%	21
Master's degree	32.35%	16
Doctorate	14.71%	6
Total	100%	50

Some refactoring tools do support this kind of customization using a general-purpose language. For example, Python Rope has a restructurings feature that allows users to write transformations using Python. We feel this is (some additional) evidence that users want this kind of flexibility in their tools. Transformation languages fit this problem domain very well as they explicitly support customizing the changes they apply. However, researchers have anecdotally noticed that transformation languages are not applied to industry code change problems [12]. This survey seeks to confirm this anecdotal notion and begin answering critical questions: What we, as researchers, do to increase the adoption of transformation languages? What are the reasons for lack of adoption of transformation languages? Are transformation languages ill-equipped, too hard to comprehend, or unneeded for the refactoring problems developers face?

Previous literature gives insight into whether they are unneeded. It shows that 1) transformation language technology provides scalable solutions to medium and large maintenance problems [2]; 2) the need for customizable, scalable solutions for modifying source code is evident in the increased interest in generating transformations and interviews with developers [10] who note that current tools only automate small portions of a largely manual refactoring effort; and 3) these last few facts indicate that a technology is required to fill this need. Transformation languages are such a technology.

An additional use case for transformation languages presents itself in support of generating transformations [13-15]. The basis

of generating transformations is to use examples of a change to generate some sequence of generic code edits that can be used to apply that same change but in a new context. However, research has to make these languages more attractive and usable to developers. One way to accomplish this is to use what transformation languages currently exist to understand why adoption has been so limited and addressed the problem appropriately. We aim to begin this process here.

3 RESEARCH METHODOLOGY

In order to answer our research questions, we conducted. a survey of 50 developers. TABLE 1. and TABLE 2. show their experience and level of education. Participants volunteered to be part of the survey by following a link they received through word of mouth. That is, we gave the link to several companies, and through those companies, it was spread through developer chatrooms and emails. We requested that those forwarding the survey only give it to other professional developers (i.e., their career is development) and not students or faculty members. The survey was left online for a total of four months to allow ample time for the link to spread and for participants to finish. Before starting the survey, participants were given two pieces of information. 1) We gave participants a link to Fowler's refactoring webpage [1] as an optional reference for some of the questions. We instructed participants to use the webpage as a reference if they needed it and to answer questions to the best of their abilities if they were unsure about anything. 2) We introduced the concept of transformation languages and standard refactoring tools by defining each similarly to how we defined them in Section 1. Since the survey compares these to one another, it is important that participants be able to differentiate the concepts. The survey data is available **via this link to Google drive**.

4 RESEARCH QUESTIONS

As stated, our research questions primarily aim to understand the current-day adoption of transformation languages in the software development life cycle. To this end, we use standard refactoring tools as a comparison point due to the similarity in purpose and function. There are four research questions; we start with RQ1 below.

RQ1: Are refactoring tools more well-known than source code transformation languages?

Standard refactoring tools are pervasive in development environments such as Visual Studio and Eclipse. This question allows us to understand how many participants are familiar with transformation languages and compare this to their familiarity with tools that can be used for a similar purpose; standard refactoring tools. We obtain the answer to RQ1 by asking developers to report which languages and refactoring tools they are familiar with. The data is in Figure 2 and Figure 1. For standard refactoring tools, Visual Studio is the most familiar to developers. This is followed by JetBrains IDEs (Resharper, Pycharm, etc.), Eclipse, Rope, Notepad++ and Refactor! Pro. We compare this result with data in Figure 1, which shows which transformation languages the participants are familiar with. We note that we allowed participants to write in languages or tools that are not listed in the survey. This mitigates the threat of us simply not naming a tool/language that looks familiar to a participant. All write-ins are contained in the figures.

The first observation we make is that the number of participants reporting that they are not familiar with any transformation languages is higher (21), compared to the number of participants that are not familiar with any standard refactoring tool (10). Further, out of all transformation tools listed, XSLT is the

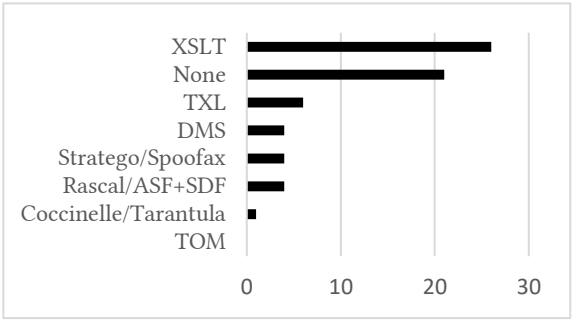

Figure 1: Participant familiarity with transformation languages. Participants were able to pick as many languages as they recognized. (n=50)

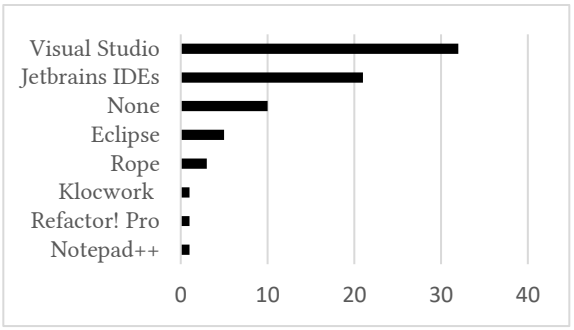

Figure 2: Participant familiarity with refactoring tools. Participants were able to pick as many tools as they recognized. (n=50)

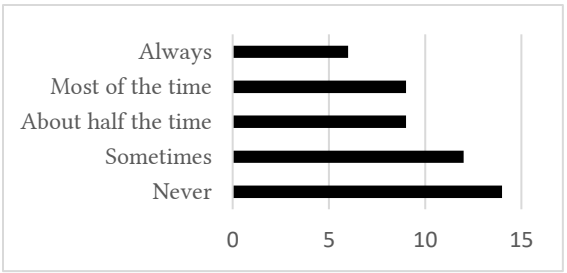

Figure 3: Frequency of refactoring tools usage. The responses show that 72% of participants used refactoring tools "Sometimes" or more, with about 28% never using them. (n=50)

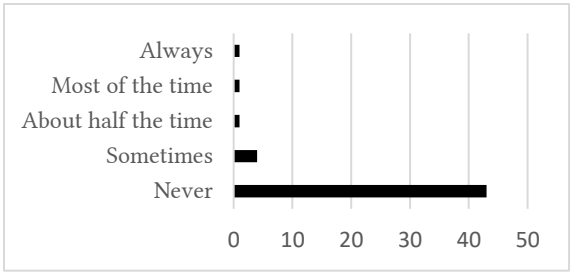

Figure 4: Frequency of transformation language usage. The responses show that only about 14% of the participants used transformation languages "Sometimes" or more, with about 86% never using them. (n=50)

most recognized. XSLT is a transformation language for XML documents. When combined with XML markup languages that are specialized for source code, such as srcML [11], it can be used to perform program transformations. There are two ways to look at this result. The first is that participants are familiar with XSLT because they have used it for other XML transformation tasks (on HTML perhaps). The second, far less likely option, is that some may have used it with an XML-based language like srcML to apply program transformations. From the data in Figure 1, it is not possible to draw conclusions.

After XSLT, the number of participants familiar with transformation languages specifically designed for source code (i.e., TXL, RASCAL) is notably lower than with standard refactoring tools. If we take the sum of all data points minus those reporting None in Figure 2 and do the same for Figure 1, then we get respectively a total of 64 and 45. Visual Studio refactoring tools are definitely used to apply automated changes to source code, but XSLT is not necessarily used as a transformation language for source code. Without XSLT, the sum of participants familiar with transformation languages (Figure 1) drops significantly to 19. For this reason, we have to determine how XSLT was used in order to draw any conclusions. To do this, we move to RQ2 and postpone the answer to RQ1 until then.

RQ2: How often do developers use transformation languages versus standard refactoring tools in their development activities?

As stated, our research questions primarily aim to understand the current-day adoption of transformation languages for applying changes to the code. We use refactorings, once again, as a comparison point to contextualize usage of transformation languages. Figure 3 contains the results for standard refactoring tools and Figure 4 contains the results for transformation languages. Comparing the two figures, we observe that, while developers use standard refactoring tools to several degrees of frequency, the vast majority of them report never using transformation languages; not even languages we may not have mentioned explicitly in the survey. In total, only 7 of the participants report at least sometimes using a transformation language. This contrasts with the 36 that reported at least sometimes using a standard refactoring tool.

The survey questions that RQ2 is concerned with allowed participants to comment when they report either sometimes or never using standard refactoring tools and/or transformation languages. We selected a subset of these comments and grouped them by similarity. Text in square brackets ([]) is added by us for contextual clarity. We start with comments made for *sometimes* or *never* using standard refactoring tools (labeled RC 1-11). There are roughly three camps. In the first, users are unaware of standard refactoring tools or are not trained to use standard refactoring tools:

RC1: "I didn't know there were tools to assist with refactoring. I've always done it manually."

RC2: [use of standard refactorings tools is] "not enforced, never motivated to, rarely/never trained on how to"

RC3: "In general I create small scripts and time usually prohibits researching/learning such tools."

In the second group, users did not trust the result of using a standard refactoring tool to apply changes and noted a preference for simple refactorings:

RC4: "When I do use refactoring, it is for refactoring. Very rarely extract method or extract variable. I never use the other refactoring tools because it is too difficult to understand what they will do. And sometimes they break the code such that it does not even compile anymore."

RC5: "The projects I work on tend to be simple enough so that refactoring tools are not needed, or the information needed to make these tools usable is not available, such as a not very well defined API where changing the name of a function could result in errors."

RC6: "For simple tasks (e.g. renaming that impacts multiple classes) I trust the refactoring tool to provide proper support and that it can handle every transformation automatically without me having to check immediately. For more complex refactorings, I always want to have a good test suite to back me up."

RC7: "Distrust of tools – would prefer to make my own code changes."

RC8: "Mostly for just renaming"

In the third group, users did not feel they needed to use standard refactoring tools due to the size or type of their code base or the efficiency/availability of the tools in their IDE.

RC9: "I work with small scripts/codebases and coworkers don't use such tools either."

RC10: "I find that the IDEs that include them are just too slow on a large industry codebase."

RC11:" Most of my work was in embedded/real-time SW development. Often OO languages were not used. Additionally, refactoring tools were not part of the culture where I worked. I cannot ever remember an instance when they were discussed."

The results and comments on standard refactoring tools imply that they are used in development, but there is still some level of unawareness, lack of trust, and some preference for using these tools on smaller/simpler problems versus larger problems amongst the developer population. Next, we look at comments for *sometimes* or *never* using transformation languages (labeled TC 1-18). We break them up into three groups. One group of developers generally report that standard refactoring tools are enough for the automatable change problems they face or that there is little motivation to try a transformation language due to lack of support in their current toolset and lack of training/education about transformation languages:

TC1: "Because IDE tooling has language semantic knowledge."

TC2: "I've only used tools to do the listed refactorings as they work well in Visual Studio"

TC3: "Other than in-built IDE support, I only use XSLT. Other languages are too complex or obscure."

TC4: "Not already present in my IDE and I don't need to do transformations that are large enough to seem worth the effort of seeking out another tool."

TC5: "Not used with the tools I use."

TC6: "Too complex (not easy to install/conFig./use)."

TC7: "The learning curve to come to grips with these tools/languages most often seems like more effort than performing a change/refactoring manually. Additionally, manual effort while tedious affords more control."

TC8: "Not much exposure to transformation languages in corporate environment"

A second group report being unaware of transformation languages or their applicability to refactoring.

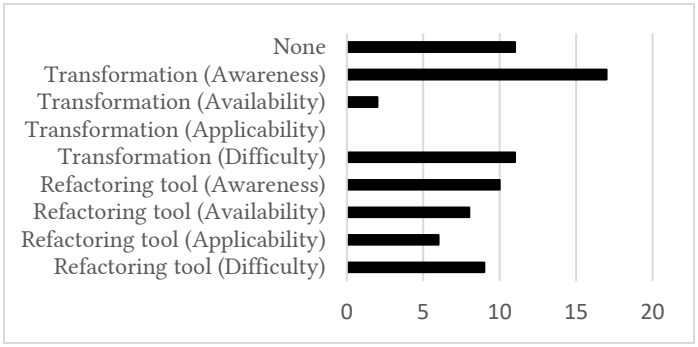

Figure 5: Perceived obstacles to using a transformation language or refactoring tool. Participants were able to make multiple selections. If they selected *none*, we prompted them to explain what they felt were obstacles.

TC9: "I don't know of any that are available to me, and that will work on my codebase."

TC 10: "No familiarity with the concept."

TC11: "I've simply never worked with one."

TC12: "Probably awareness. I have never heard of most of these transformation languages. I have had to use XSLT in the past, but the language is very complicated and it takes a lot of effort to use, especially since XSLT must be written in XML. XSLT is not something I go to unless required by a job."

TC13: "I've never learned (or even been exposed to) such a language."

And a third group worry about the result of applying a transformation language or note use cases to which a transformation language might apply, but ultimately state that they rarely or never see those cases.

TC14: "Generally, simple refactorings such as a renaming classes\methods\etc, extracting methods, moving classes, etc are done using IDE tooling. So you never really consider the underlying tech that's actually performing the refactoring. For larger refactorings or architecture changes, we manually make changes code changes. Occasionally we'll write scripts that modify our code for us using scripting languages like bash, cmd, python, ruby, etc. But those are few and far in between."

TC15: "I'm worried it would screw up all the code."

TC16: "Never had the case where many changes of the same type needed to be made which is why I would use a transformation language."

TC17: "I haven't encountered a problem that I'd really need a large scale fix for something like this. Although if I had a tool like this built in I might use it for searching more then anything else."

TC18: "Tools are good enough, a language would probably help for a serious refactoring work that may scale up to full project transformation"

TC19:" I typically work with a small codebase where any refactoring takes a small amount of time. If I worked with a much larger project, then I would spend the time to write a transformation rather than manually refactoring."

Comparing both comments and data from the figures, more participants are unaware of or feel warier of transformation languages than standard refactoring tools and generally seem to favor the idea of a refactoring tool than a language for applying code changes. TC Group 1 saw little motivation to use a more flexible tool given their experiences or feel that languages are too complex; TC Group 2 felt an overall lack of training in and/or lack of awareness of transformation languages, which perhaps makes it hard for them to gauge whether they would use one or even why they might use one; TC Group 3 worried about the resulting code post-transformation or felt there may be use cases, but had never run into one.

Some interesting comments to point out are TC 4, 14, 16, 17,18, and 19. TC 14 makes an observation very similar to an observation made in Kim et al.'s study [10]. Specifically, they observed that larger architectural refactorings are done manually, while automated refactorings tend to be smaller. The observation in [10] notes that automated refactorings tend to be small/low level and applied as part of a more substantial, mostly manual architectural refactoring. TC 4, 16, 17, 18, and 19 support this line of reasoning; implying or outright stating in their comments that they might use a transformation language for large-scale changes, meaning that they see this as a niche that standard refactoring tools are not currently filling.

The answer to RQ2 is that when developers do automatically apply changes to their code, they prefer using standard refactoring tools or manually applying changes more often than transformation languages. Additionally, we have enough information to answer RQ1. Given that 43 developers report never using a transformation language to refactor, that only leaves 7 developers that report at least sometimes using a transformation language for refactoring. This means that a maximum of 7 developers might have used XSLT for this purpose. The answer to RQ1 is that developers surveyed are more familiar with standard refactoring tools than transformation languages.

RQ3: What do developers perceive as obstacles to the adoption of transformation languages?

At this point, we understand that participants are more familiar with standard refactoring tools and tend to prefer them over transformation languages for applying refactorings. The purpose of RQ3 is to begin understanding more about what participants feel are barriers-of-entry to using transformation languages and, for comparison, standard refactoring tools. Since not all developers are familiar with transformation languages, answering this question was optional. We answer this question using Figure 5. As part of the survey, we list a generic set of potential reasons for why a developer might avoid using a transformation language or standard refactoring tool. The generic reasons we provided are as follows:

- Difficulty: language/tool is too hard to use
- Applicability: language/tool does not implement, or cannot implement, required refactoring
- Availability: language/tool does not work on required language, or in required IDE
- Awareness: unaware of the existence of languages/tools for refactoring.

If the participant felt that none of the reasons listed are obstacles, we prompted them to list their own. We also allowed participants to select more than one reason. With respect to standard refactoring tools, participants are nearly evenly split between difficulty (9), applicability (6), availability (8), and awareness (10). For transformation languages, there is more variance: difficulty (11), applicability (0), availability (2), and awareness (17). Finally, there are a total of 11 users that felt none of the reasons we provided are obstacles. When asked to expand on what they meant, only 6 responded. Two felt that transformation languages are not worth the effort, given their

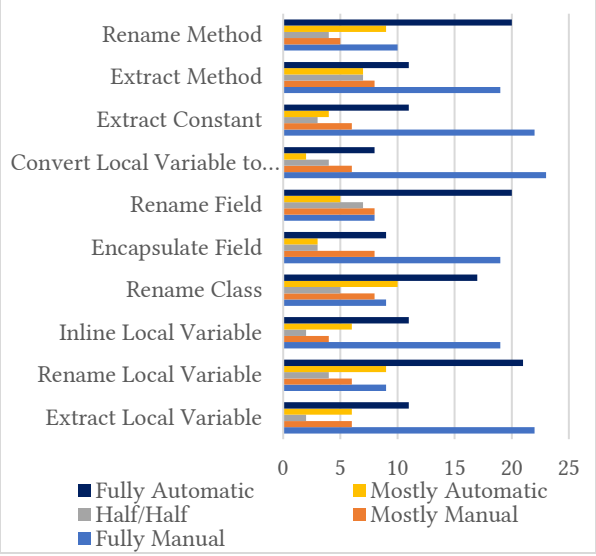

Figure 6: Frequency at which participants currently automatically or manually apply common refactorings.

experience or position. One is indifferent; seeing no obstacles. The last feels lack of support is the main obstacle (Labeled C1-4). Text in square brackets ([]) is added by us for contextual clarity.

C1: [With respect to transformation languages,] "Effort involved is not worth the benefit for most tasks. Good programming practices also limit the effectiveness of these tools."

C2: [Transformation languages are] "Not necessary for the kinds of code I write."

C3:" Nothing? I'm too indifferent to most things in life to really feel like anything is specifically an obstacle. I just work around things and don't care."

C4:"Languages not implementing [transformation] language services for existing tools."

We did not report the other two written comments because they did not include enough information to understand what the participant was trying to say. There are some insights we can derive from what we know so far. First, participants did feel that transformation languages are powerful enough to solve their refactoring problems (0 voted for applicability as an obstacle).

Next, the most frequently selected obstacles preventing developers from adopting transformation languages are: Lack of awareness (17) and difficulty (11), respectively. The same two are the two top obstacles for refactoring tools: Awareness (10) and difficult (9). These two are perceived problems for both, but more so for transformation languages, particularly regarding awareness. Applicability (0) and Availability (2) are generally not considered obstacles for transformation languages compared to refactoring tools: Applicability (6), Availability (8).

We did not ask for comments when users selected difficulty or awareness as an obstacle, but for some additional context, we looked at comments those users left on a previous question, where they explained why they sometimes or never use transformation languages (i.e., TC1-18 above). There is a total of 26 responses. 11 users generally report lack of awareness of transformation languages, 8 users report that either they are unsure of whether learning one would be worth it, or they did not have sufficient motivation to learn one; 5 users report that they are worried about the complexity/difficulty of using a transformation language, and two users left no comment.

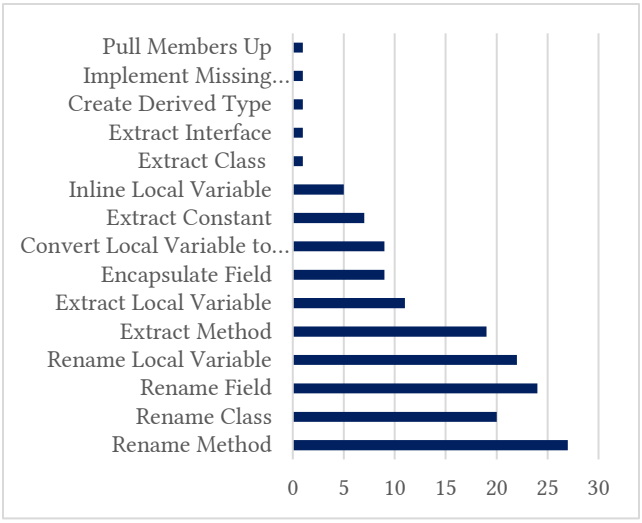

Figure 7: Types of refactorings that participants most want to see automated. Participants could select more than one. Write-ins were added this figure.

11 participants (no intersection with the previous twenty 6) selected *None*. 4 of them gave a thorough enough response for us to report above (C1-4). To help us gain further insight, we looked at comments made in previous questions by those who selected None (e.g., TC1-18 above). 3 of these participants report using transformation languages and felt there are no problems. 5 of them felt there is simply no need for them and 3 reported a lack of opportunity or motivation to try transformation languages. That is, 8 of the 11 who voted *None* felt that there is no obstacle to adoption; they simply have no need or motivation to try transformation languages and, even if one were available and easy-to-use, it appears that they do not want use one. By contrast, one person who voted *None* felt there is no need for refactoring tools.

Most of the users who voted for difficulty as an obstacle have very similar comments to 11 of the users who voted *None*. One explanation for this similarity may be that while those who voted *None* feel there is zero need for transformation languages in their tasks, those who voted for difficulty felt that learning a transformation language is currently too much cost for the payoff. It might be they would reconsider if the perceived difficulty is minimized or the perceived benefit is increased.

The answer to RQ3 is as follows: More participants reported lack of awareness of transformation languages than refactoring tools; awareness was the most significant obstacle. More also report difficulty as a problem they attributed to transformation languages than refactoring tools, and more felt that there is nothing that will convince them (i.e., no obstacles to overcome) to use transformation languages compared to refactoring tools.

TABLE 3. TOTAL PREFERENCE FOR MANUAL OR AUTOMATIC APPLICATION OF REFACTORINGS ACROSS ALL REFACTORING TYPES.

Fully Automatic	Mostly Automatic	Half/Half	Mostly Manual	Fully Manual
139	61	41	65	160

RQ4: What do the standard refactoring practices of our participants look like?

In RQ3, we tried to understand what users felt are obstacles to the adoption of transformation languages. In RQ4, we answer the same question except use answers to previous RQs in conjunction with data about refactoring habits to help us understand how these

habits might affect participant perception of transformation languages. To do this, we use data from Negra et al. [3].

First, we generated a list of common refactorings based on data from Negara's study. We then ask our participants to perform the following task: *Please indicate the extent to which you manually or automatically apply the following refactorings.* The results of the question are given in Figure 6 and TABLE 3. They show that participants generally prefer either fully automating a refactoring or fully applying a refactoring manually rather than going halfway or even most of the way in either direction. They also show a nearly even split between the preference for manually applying refactorings and automatically applying them. We conclude that, among the participants, anything more complex than rename operations is more likely to be manually applied.

Second, using the same list of common refactorings, we asked participants to perform the following task: *Please indicate which refactorings you consider most important to automate.* We begin by comparing Figure 7 to Figure 6. At first glance, it is easy to see the rename refactorings are still a high priority. Looking at *Extract Method*, we notice that in Figure 6, 11 developers report automating it. However, in Figure 7, 19 developers report wanting to see it automated. This is the largest increase among any frequency between the two figures. Participants primarily want rename-type refactorings automated. Only two non-rename refactorings that saw an increase between Figure 6 and Figure 7: Extract Method and Convert Local Variable To Field. *RQ4's answer is that participants are most comfortable with automating rename refactorings; they preferred to manually apply most other more complex refactorings except Extract Method and Convert Local Variable To Field.*

5 IMPLICATIONS

We now discuss the obstacles derived from this study and then ways to remediate these obstacles.

5.1 Obstacles to Adopting Transformation Languages

Transformation languages provide no significant benefit for the refactorings that developers are comfortable automating; developers do not trust complex automated changes that may benefit from a transformation language

This is most clearly confirmed in RQ3/RQ4 and seen in Figure 6 and Figure 7, where the top three refactorings developers want to automate are renames. Renames are high-level refactorings, i.e., they only affect the signature of the code element (e.g., class, method, field) and they do not change its implementation. Rename refactorings are very well supported by existing refactoring tools and would not become simpler to apply with a transformation language. Tasks with higher complexity, such as the *new* operator example in Section II, benefit more from a transformation language, but previous work, as well as data from Section 4 , imply that developers are uncomfortable with medium or large, fully-automatically applied changes [10, 16]; they prefer to be allowed to modify and oversee these changes in real time.

Transformation languages are perceived as difficult

11 developers felt that transformation langauges are too difficult to use, but we found that these users also reported never having used a transformation languge for refactoring. We compared their comments from RQ1/RQ2 to the comments of the 11 developers that had reported there being no obstacle to adopting transformation languages and found that they were very similar: Transformation languages are not part of their IDE, they have no experience with them, they are most comfortable applying simpler refactorings automatically. This indicates that difficulty, in this case, maybe a cost-benefit measure; participants assume that transformation languages will be too hard to install/config/learn/apply compared to the potential benefit they provide. This is supported by the previous factor: if developers only use/trust simpler automated refactorings, there is no reason to undergo the difficulty of learning a language—they will do more complex refactorings manually.

Exposure to transformation languages is low

Using RQ1, and RQ2, we observe that familiarity with transformation languages is lower than with refactoring tools. Additionally, in RQ3 we learned that, among developers who felt there were obstacles to adopting transformation languages, awareness was the highest reported obstacle of all obstacles for refactoring tools and transformation languages.

5.2 Remediating these Obstacles

We now prescribe avenues of research which the authors think will help encourage greater adoption of transformation languages for refactoring problems. We reiterate that industry is looking for a solution to these refactoring problems; developers want safe, customizable, generalizable refactoring engines; transformation languages are appropriate for this.

Increase developer confidence in refactoring and transformation tools

Developer confidence is an issue for both refactoring tools and transformation languages. Researchers are already working on safety of refactoring tools [17], which will help encourage their use by developers. Research on transformation languages must also demonstrate that methods to ensure refactoring safety will also work on languages or extend/create new methods. *Guaranteeing safety will reduce cost of adoption and increase the attractiveness of refactoring engines.*

Study how user-friendly modern transformation languages are and improve on them if required

Many transformation languages already exist for various purposes. There is not much research into their usability, however. Developers have a difficult job as it stands, and typically need to be proficient in several languages. If we are going to say that they should know more, then research should be able to argue how transformation languages should be practically applied and how it can most effectively help developers perform refactoring tasks. This means cooperating with industry to design languages that are as easy to learn as possible without sacrificing expressiveness and applicability. *A visible, well-designed language will lower the barrier of entry and perhaps convince developers to experiment.*

Study the application of transformation languages on larger change-tasks

Combining RQ3/RQ4, participant comments (Section 4), and previous work [10], one consistent issue we see is that larger refactoring tasks (e.g., api migration, architectural/design changes) tend to be applied manually. Research should help make the case that larger, typically manually applied refactorings can be done more easily, more quickly, and more safely using a language. Previous work already shows that it is possible [2, 18], but more data is required to quantify how much better languages will perform with respect to maintainability (e.g., is the changed code safe and comprehendible?) and acceptableness (e.g., will developers accept the changes?) after an applied change. *Safe, well-designed transformation languages that can clearly solve a set of problems developers face will stand the greatest chance of being adopted and used.*

6 THREATS TO VALIDITY

One potential threat is the bias of our sample due to distribution via email. We mitigated this by asking the initial batch

of respondents to forward this survey to other colleagues and professional developers they know. We also sent the survey to several companies across several states and countries. Additionally, the IP addresses recorded by Qualtrics indicate that respondents did not cluster significantly in terms of geography; respondents were from places such as California, the Carolinas, Ohio, New York, Michigan, Colorado, Florida, Washington and Canada. Also, some participants were employed at companies such as Progressive, ABB, Microsoft and Google.

7 RELATED WORK

One of the earliest papers to discuss the static nature of standard refactoring tools is by Kniessel and Koch [19], where they propose composing simple refactoring operations into larger, more complex refactoring operations. Several studies investigate the use of refactoring tools by developers and conclude that refactoring tools are underused to begin with [3-5]. Specifically, Negara et al. [3] did a study on manual and automated refactorings; comparing, among other things, the frequency, proportion, and the most popular of each type. Murphy et al. differentiate floss refactoring and root-canal refactoring [5]. Vakilian et al. [18] study automated refactorings and find similar patterns as Murphy. Moser et al. [20] performs a case study and find that refactoring increases quality and improves productivity. Kim et al. [10] did an empirical study at Microsoft. In the study, 46% of developers agreed that "refactorings supported by automated tools differ from the kind they perform manually." Previous studies on developer perception of refactorings focus on code smells [21, 22].

Several mature transformation tools exist. Rascal [6] was designed by Paul Klint. The authors of Rascal explicitly discuss and motivate its application to refactoring. Spoofax [23] is designed by Elco Visser and has seen use in refactoring previously [9]. Li and Thompson [12] present a transformation language using an Erlang tool called Wrangler to perform refactorings. Coccinelle supports refactoring on Linux [24, 25]. DMS is an infrastructure for program transformation; used for large scale migration previously [26]. srcML is a markup language for source code, used to modify code using various transformation languages in the past [2, 8, 27].

8 CONCLUSIONS

The goal of this study was to investigate the use of transformation languages in industry. We chose to use refactorings as the point of comparison due to the amount of data about refactorings, categories of refactorings, and refactoring tools. We argued that transformation languages can serve a useful purpose for refactoring. Unfortunately, there was very little previous research on whether transformation languages are used in industry for refactoring and how they can be improved to more fully support developers. To investigate these questions, we surveyed 50 professional developers, discussed obstacles to the adoption of transformation languages and paths to remediation.

REFERENCES

[1] M. Fowler, Refactoring: improving the design of existing code: Addison-Wesley Longman Publishing Co., Inc., 1999.

[2] M. L. Collard, J. I. Maletic, and B. P. Robinson, "A lightweight transformational approach to support large scale adaptive changes," presented at the Proceedings of the 2010 IEEE International Conference on Software Maintenance, 2010.

[3] S. Negara, N. Chen, M. Vakilian, R. E. Johnson, and D. Dig, "A comparative study of manual and automated refactorings," presented at the Proceedings of the 27th European conference on Object-Oriented Programming, Montpellier, France, 2013.

[4] E. Murphy-Hill, C. Parnin, and A. P. Black, "How we refactor, and how we know it," presented at the Proceedings of the 31st International Conference on Software Engineering, 2009.

[5] E. Murphy-Hill and A. P. Black, "Refactoring Tools: Fitness for Purpose," IEEE Softw., vol. 25, pp. 38-44, 2008.

[6] P. Klint, T. v. d. Storm, and J. Vinju, "RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation," presented at the Proceedings of the 2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation, 2009.

[7] J. R. Cordy, "The TXL source transformation language," Sci. Comput. Program., vol. 61, pp. 190-210, 2006.

[8] C. D. Newman, "A SOURCE CODE TRANSFORMATION LANGUAGE TO SUPPORT SOFTWARE EVOLUTION," Kent State University, 2017.

[9] M. D. Jong and E. Visser, "Implementing Refactorings in the Spoofax Language Workbench," SERG Technical Report, pp. 1-19, 2013.

[10] M. Kim, T. Zimmermann, and N. Nagappan, "An Empirical Study of Refactoring Challenges and Benefits at Microsoft," IEEE Transactions on Software Engineering., vol. 40, pp. 633-649, 2014.

[11] M. L. Collard, M. J. Decker, and J. I. Maletic, "srcML: An Infrastructure for the Exploration, Analysis, and Manipulation of Source Code: A Tool Demonstration," presented at the Proceedings of the 2013 IEEE International Conference on Software Maintenance, 2013.

[12] H. Li and S. Thompson, "Let's make refactoring tools user-extensible!," presented at the Proceedings of the Fifth Workshop on Refactoring Tools, Rapperswil, Switzerland, 2012.

[13] N. Meng, M. Kim, and K. S. McKinley, "LASE: locating and applying systematic edits by learning from examples," presented at the Proceedings of the 2013 International Conference on Software Engineering, San Francisco, CA, USA, 2013.

[14] R. Rolim, G. Soares, L. D'Antoni, O. Polozov, S. Gulwani, R. Gheyi, et al., "Learning syntactic program transformations from examples," presented at the Proceedings of the 39th International Conference on Software Engineering, Buenos Aires, Argentina, 2017.

[15] N. Meng, M. Kim, and K. S. McKinley, "Sydit: creating and applying a program transformation from an example," presented at the Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering, Szeged, Hungary, 2011.

[16] M. W. Mkaouer, M. Kessentini, S. Bechikh, K. Deb, and M. Ó Cinnéide, "Recommendation system for software refactoring using innovization and interactive dynamic optimization," in Proceedings of the 29th ACM/IEEE international conference on Automated software engineering, 2014, pp. 331-336.

[17] T. Mens, S. Demeyer, and D. Janssens, "Formalising Behaviour Preserving Program Transformations," Berlin, Heidelberg, 2002, pp. 286-301.

[18] M. Vakilian, N. Chen, S. Negara, B. A. Rajkumar, B. P. Bailey, and R. E. Johnson, "Use, disuse, and misuse of automated refactorings," presented at the Proceedings of the 34th International Conference on Software Engineering, Zurich, Switzerland, 2012.

[19] G. Kniesel and H. Koch, "Static composition of refactorings," Sci. Comput. Program., vol. 52, pp. 9-51, 2004.

[20] R. Moser, P. Abrahamsson, W. Pedrycz, A. Sillitti, and G. Succi, "A Case Study on the Impact of Refactoring on Quality and Productivity in an Agile Team," in Balancing Agility and Formalism in Software Engineering, M. Bertrand, R. N. Jerzy, and W. Bartosz, Eds., ed: Springer-Verlag, 2008, pp. 252-266.

[21] F. Palomba, G. Bavota, M. D. Penta, R. Oliveto, and A. D. Lucia, "Do They Really Smell Bad? A Study on Developers' Perception of Bad Code Smells," presented at the Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution, 2014.

[22] A. F. Yamashita and L. Moonen, "Do developers care about code smells? An exploratory survey," 2013 20th Working Conference on Reverse Engineering (WCRE), pp. 242-251, 2013.

[23] L. C. L. Kats and E. Visser, "The spoofax language workbench: rules for declarative specification of languages and IDEs," SIGPLAN Not., vol. 45, pp. 444-463, 2010.

[24] J. Lawall, G. Muller, and R. Urunuela, Tarantula: Killing Driver Bugs Before They Hatch, 2005.

[25] Y. Padioleau, J. L. Lawall, and G. Muller, "SmPL: A Domain-Specific Language for Specifying Collateral Evolutions in Linux Device Drivers," Electron. Notes Theor. Comput. Sci., vol. 166, pp. 47-62, 2007.

[26] R. L. Akers, I. D. Baxter, and M. Mehlich, "Invited application paper: re-engineering C++ components via automatic program transformation," presented at the Proceedings of the 2004 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation, Verona, Italy, 2004.

[27] V. Augustine, "Automating adaptive maintenance changes with SrcML and LINQ," presented at the Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, Cary, North Carolina, 2012.