Does the Introduction of Lambda Expressions Improve the Comprehension of Java Programs?

Walter Lucas, Rodrigo Bonifácio, Edna Dias Canedo, Diego Marcílio, and Fernanda Lima
Computer Science Department, University of Brasília (UnB)
Brasília, Brazil

ABSTRACT

Background: The Java programming language version eighth introduced a number of features that encourage the functional style of programming, including the support for lambda expressions and the Stream API. Currently, there is a common wisdom that refactoring a legacy code to introduce lambda expressions, besides other potential benefits, simplifies the code and improves program comprehension. **Aims**: The purpose of this paper is to investigate this belief, conducting an in depth study to evaluate the effect of introducing lambda expressions on program comprehension. **Method**: We conduct this research using a mixed-method study. First, we quantitatively analyze 66 pairs of real code snippets, where each pair corresponds to the body of a method before and after the introduction of lambda expressions. We computed two metrics related to source code complexity (number of lines of code and cyclomatic complexity) and two metrics that estimate the readability of the source code. Second, we conduct a survey with practitioners to collect their perceptions about the benefits on program comprehension, with the introduction of lambda expressions. The practitioners evaluate a number between three and six pairs of code snippets, to answer questions about possible improvements. **Results**: We found contradictory results in our research. Based on the quantitative assessment, we could not find evidences that the introduction of lambda expressions improves software readability—one of the components of program comprehension. Differently, our findings of the qualitative assessment suggest that the introduction of lambda expression improves program comprehension. **Implications**: We argue in this paper that one can improve program comprehension when she applies particular transformations to introduce lambda expressions (e.g., replacing anonymous inner classes by lambda expressions). In addition, the opinion of the participants shine the opportunities in which a transformation for introducing lambda might be advantageous. This might support the implementation of effective tools for automatic program transformations. Finally, our results suggest that state-of-the-art models for estimating program readability are not helpful to capture the benefits of a program transformation to introduce lambda expressions.

SBES 2019, September 23–27, 2019, Salvador, Brazil

ACM ISBN 978-1-4503-7651-8/19/09…$15.00
https://doi.org/10.1145/3350768.3350791

CCS CONCEPTS

• **Software and its engineering → Empirical software validation**; **Software evolution**; **Maintaining software**;

KEYWORDS

Program Comprehension, Java Lambda Expressions, Empirical Studies

ACM Reference Format:
Walter Lucas, Rodrigo Bonifácio, Edna Dias Canedo, Diego Marcílio, and Fernanda Lima. 2019. Does the Introduction of Lambda Expressions Improve the Comprehension of Java Programs?. In *XXXIII Brazilian Symposium on Software Engineering (SBES 2019), September 23–27, 2019, Salvador, Brazil.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3350768.3350791

1 INTRODUCTION

Different factors motivate *software maintenance* efforts, including business evolution, requirements changes, architectural transitioning (towards the use of microservices or cloud environments, for instance), and the need for quality improvements (such as the use of more recent APIs or program refactoring) [19]. Low-level technical aspects might also motivate a software evolution effort. For instance, the evolution of a programming language, which often introduces new programming language constructs (such as Generics in Java 5 and Lambda Expressions in Java 8), might trigger maintenance efforts of "legacy systems". This kind of *software rejuvenation* is important to reduce the coexistence of different idioms in software programs, mostly because developers often start using new programming language constructs during the development of new features, while keeping obsolete constructs in the existing legacy code.

Without a strategic decision, obsolete constructs are rarely removed from older parts of a program [15]. Although there are examples of "new" programming language constructs that practitioners have rapidly understood, embraced, and adopted (e.g., Java Generics and Java Annotations) [16], the introduction of lambda expressions in Java does not show the same appeal. For instance, Dantas et al. report the results of sending rejuvenation patches to open-source programs with a set of Java program transformations [6]. Curiously, the introduction of lambda expressions to replace (a) anonymous inner classes and (b) existing *for loops* were the types of transformations with the lowest acceptance rate. Since there are claims that the use of Java lambda expressions improves program comprehension [11], these previous results motivate further investigation to better understand the reasons for that low patch acceptance rates. Our goal in this paper is to investigate this issue by conducting a comprehensive assessment about (a) whether or not the use of lambda expressions improves program comprehension and (b) the

most suitable situations to refactor a code to introduce Lambda Expressions.

We found conflicting results. Based on a quantitative investigation using state-of-the-art metrics for program comprehension, our findings suggest that Java lambda expressions **do not improve** program comprehension. Differently, based on the results of a survey with practitioners, we identified several scenarios where a *refactoring* to introduce lambda expressions **actually improves** program comprehension. For instance, qualitative findings reveal that replacing anonymous inner classes with small method bodies into lambda expressions brings benefits for overall code comprehension.

Our study has several implications. First, existing claims that the adoption of lambda expressions benefits program comprehension only hold for specific cases—even though the Java support for the functional style was a long waited feature. Our results provide some guidance about the scenarios where the introduction of a lambda expression give benefits on program comprehension. Second, we give evidence that state-of-the-art metrics for program comprehension fail to identify whether a given source-code transformation leads to an improvement on the source code. Although here we focus on transformations that introduce lambda expressions, these metrics might also fail in other scenarios as well.

2 BACKGROUND AND RELATED WORK

Program comprehension is a fundamental attribute that allows software maintenance and supports the evolution of a software [25]. Understanding existing software enables maintainers to successfully evolve functionality and/or integrate improvements, for every type of change that is commonly associated with software maintenance and evolution, including adaptive, perfective, and corrective modifications [25]. The challenges to understand a software are due to several factors, including that large programs are often maintained by developers with different skills and using different practices [22]. Moreover, in many cases, the source code is the only reference that is available and up to date [22], though poor design and lack of good programming practices hinder program comprehension [23].

The practices developers use for understanding a software vary, and often depend on a specific motivation (e.g., documenting part of a system, fixing a bug, implementing a new feature). Nonetheless, according to Tilley and Paul, "programmers make use of domain knowledge, programming knowledge, and comprehension strategies when attempting to understand a program" [23]. Therefore, program comprehension uses existing knowledge to acquire new knowledge, in order to build a mental model of the software that might help developers to accomplish a specific task [25]. While it is true that the skills and experiences of a software developer are relevant when he / she wants to understand a software, it has been reported that a set of recommended practices (such as the use of programming idioms and code formatting tools, design patterns, and refactoring) might also support program comprehension, in particular when using a bottom-up strategy [17]. Conversely, the use of some *obscure programming constructs*, named atoms of confusion, for instance, increases the rate of source code misunderstandings [10].

Although many features of a software might impact program comprehension, in this paper we are particularly interested in aspects related to the quality of the source code that might either facilitate or hinder program understanding [22]. To investigate similar issues, several research studies have explored the use of models for estimating the readability of the source code (e.g., [4, 18, 21]), which directly affect program comprehension. In addition, previous research has already investigated the impact on software readability with the use of coding practices [7, 10]. Our work builds upon these previous works, using existing models for estimating software readability [4, 18], and procedures to qualitatively assess the preference of practitioners, when considering sets of code snippets [7]. We apply these results of previous research in a different scenario, related to the introduction of lambda expressions into Java legacy code.

Lambda expressions were introduced in Java 8 as a means to support functional programming [24]. They allow functionality to be treated as a method argument, code, or data [1], enabling what the literature calls behavior parameterization [14]. Lambda expressions are suitable for parallel constructs and can also substitute anonymous classes with less code (shorter syntax) [1]. In practice, Java lambda expressions is a shorter way of expressing instances of single abstract method interfaces. Another use is to enable the use of internal iterators within the Java 8 collections API that takes a lambda expression as an argument [11]. Functional operations like *map* and *filter* can be applied together with *streams* as an alternative way to iterate, filter, and extract data from a collection [14]. For instance, consider the code snippets on Listing 1 and Listing 2, based on an implementation of the *101Companies* problem domain [9]. In this example, the goal is to filter the employees of a department that have a salary greater than a given value. In the first snippet, the code uses an implementation without the language features of Java 8. In the second, the implementation uses a lambda expression as an argument to the `filter` method of the Java 8 *stream* API.

```
public List<Employee> employeeWithHighSalaries(double salary) {
  List<Employee> res = new ArrayList<>();
  for(Employee e: employees) {
    if(e.getSalary() > salary) res.add(e);
  }
  return res;
}
```

Listing 1: Filtering employees with high salaries (approach previous to Java 8)

```
public List<Employee> employeeWithHighSalaries(double salary) {
  return employees.stream()
                  .filter(e -> e.getSalary() > salary)
                  .collect(Collectors.toList());
}
```

Listing 2: Filtering employees with high salaries (approach using lambda expressions and the Java 8 stream API)

Previous research on Java lambda expressions has focus on automatic methods for refactoring legacy code to "make the code more succinct and readable" using lambda expressions [6, 11]—in particular situations that we can, for instance, replace either an *anonymous inner class* or a *loop over a collection* by statements involving lambda expressions. Other approaches recommend transformations that introduce lambda expressions to remove duplicated code [24] and to

correctly use parallel features of Java 8 [12]. In addition, Mazinanian et al. present a comprehensive study on the adoption of Java lambda expressions [14], in order to understand the motivations that lead Java developers to adopt the functional style of thinking in Java. The authors contribute with an infrastructure that collects information about the adoption of lambda expressions in Java, publishing a large dataset with more than 100 000 real usage scenarios. We use this dataset in our research, to understand the benefits on program comprehension with the adoption of Java lambda expressions.

At first, the use of lambda expressions, due to its conciseness, yields a more succinct and readable code [6, 11]. However, this is not always the case, as Dantas et al. [6] produced automated refactorings for iterating on collections that were not perceived as more comprehensible. We aim to investigate further which scenarios are benefited from the introduction of lambda expressions. To the best of our knowledge, previous research did not investigate the assumption that the use of lambda expressions actually lead to benefits on program comprehension.

3 STUDY SETTINGS

The general goal of this research is to investigate the benefits on code comprehension after refactoring Java methods to introduce lambda expressions, and, thus, answering the research questions we present in Section 3.1. To this end, we conduct a mixed-method study. First, we carry out a quantitative assessment of 66 pairs of code snippets, using state-of-the-art models for measuring software comprehension (see Section 3.2). Each pair corresponds to a method body *before* and *after* introducing lambda expressions. The second investigation is based on a qualitative study (survey) that we conduct with two distinct populations (a) experience developers and (b) undergraduate students. During the survey, the participants answered questions that also aim to compare the code before and after the use of lambda expressions.

3.1 Research Questions

Considering the general goal of our research, we address several questions in our study, including

(Q1) *Does the use of lambda expressions improve program comprehension?*
(Q2) *Does the introduction of lambda expressions reduce source code complexity?*
(Q3) *What are the most suitable situations to refactor a code to introduce lambda expressions in Java?*
(Q4) *How do practitioners and students evaluate the effect of introducing a lambda expression into a legacy code?*

We conduct this research using an iterative approach, and after answering a given question, new sub-questions and hypothesis emerged. For instance, during our research, we also investigate whether or not the reduction in the size of a code snippet, after introducing a lambda expression, influences on the perception of the participants about the quality of a transformation.

3.2 Metrics of the Quantitative Study

We measure the *complexity* of a code snippet using two metrics: number of *source lines of code* (SLOC) and *cyclomatic complexity* (CC). Both metrics have been used in a number of studies [2, 13, 20]. In addition, we use two models to estimate and compare the readability of each pair of the code snippets considered in our research. Readability is one of the aspects used for assessing program comprehension, and hereafter both terms are used interchangeably. The first model we use to estimate program comprehension is based on the work of Buse and Weimer [4]. It estimates the comprehensibility of a code snippet considering a regression model that takes as input several features, including several properties of a snippet, such as the length of each line of code, and the number and length of identifiers.

The second model was proposed by Posnett et al. [18], which builds upon the Buse and Weimer model, though considering a smaller number of source code characteristics. Based on this model, we can estimate the readability of a code snippet using Eq. (1) and Eq. (2).

$$E(X) = \frac{1}{1 + e^{-Z(X)}} \quad (1)$$

$$Z(X) = 8.87 + 0.40\ L(X) - 0.033\ V(X) - 1.5\ H(X) \quad (2)$$

That is, in the Posnett et al. model, we calculate program comprehension using three main components: the number of lines of a code snippet ($L(X)$), the volume of a code snippet ($V(X)$), and the entropy ($H(X)$) of a code snippet. The volume of a code snippet X is given by $V(X) = N(X)log_2 n(X)$, where $N(X)$ is the program length of the code snippet and $n(x)$ is the program vocabulary. These measures are defined as

- **Program Length ($N(X)$)** is given by $N(X) = N1(X) + N2(X)$, where $N1(X)$ is the number of operators and $N2(X)$ is the number of operands of a code snippet.
- **Program Vocabulary ($n(X)$)** is computed using the formula $n(X) = n1(X) + n2(X)$, where $n1(X)$ is the number of unique operators and $n2(X)$ is the number of unique operands of a code snippet.

The entropy of a document X (in our case a code snippet) is given by Eq (3), where x_i is a token in X, $count(x_i)$ is the number of occurrences of x_i in the document X, and $p(x_i)$ is given by Eq (4). The entropy ($H(X)$) in our context estimates the degree of disorder of the source code.

$$H(X) = -\sum_{i=1}^{n} p(x_i)\ log_2\ p(x_i) \quad (3)$$

$$p(x_i) = \frac{count(x)}{\sum_{j=1}^{n} count(x_j)} \quad (4)$$

We use an existing tool[1] of Buse and Weimer to estimate the comprehensibility of the code snippets using their model [4]. We developed a tool to automate the computation of the Posnett et al. comprehensibility model [18].[2] We executed these computations for all pairs of code snippets that we collected using the procedures detailed in what follows.

[1] http://www.arrestedcomputing.com/readability/
[2] https://github.com/rbonifacio/program-comprehension-metrics

3.3 Selection of the Code Snippets

We use MinerWebApp[3] to identify code snippets candidates to our research. This tool monitors the adoption of Java lambda expressions in open-source projects hosted on GitHub, and has been used in previous research on the adoption of lambda expressions [14]. MinerWebApp connects to the git source code repository of projects, in order to identify and classify the use of lambda expressions in Java code. This decision of using MinerWebApp simplified our process of collecting real usage scenarios of lambda expressions. MinerWebApp classifies the occurrences of lambda expressions into three categories:

- *New method*: When a new method containing lambda expressions is added to an existing project class;
- *New class*: When a new class is added to the project, comprising methods with *lambda expressions*;
- *Existing method*: When a lambda expression is introduced into an existing method.

From the possibility of collecting the examples directly from an existing *commit*, we randomly select MinerWebApp projects in which there were *lambda expressions*. Of the selected projects, 59 code snippets were extracted (directly from the GitHub page of the corresponding commit). The extraction considered the code snippets of the third category (*Existing method*) exclusively. We also collected 29 code snippets of *refactoring scenarios* to introduce lambda expressions [6]. In total, 88 code snippets from 22 projects were selected, including snippets from projects Elastic Search, Spring Framework, and Eclipse Foundation. We manually reviewed these code snippets and removed 22 pairs of code that does not correspond to a refactoring or that already had lambda expressions in the code snippet before the transformation, building a final dataset with 66 pairs of code snippets.

All procedures to collect and characterize the code snippets from GitHub pages have been automated, using a crawler and additional scripts for computing source code metrics (Figure 1 shows an overview of the approach). To foster the reproducibility of our study, our crawler expects as input a CSV file, where each line specifies the project, the URL of the commit, the start and end lines of the code snippet method, and the type of the refactoring (e.g., anonymous inner class to lambda expression, for each statements to a recursive pattern using lambda expressions). After downloading the pairs of code snippets, we run several scripts to compute the source code metrics and to estimate the readability models. Lastly, the code snippets and the results of the metric calculations are stored in a database—for helping with the procedures to conduct the surveys and for further statistical analysis.

3.4 Procedures of the Qualitative Study

Regarding the qualitative study, we conduct the research using an approach based on the work of dos Santos and Gerosa [7]. That is, we designed an online survey that allows the participants to evaluate pairs of code snippets. We conducted our survey in two phases. In the first, we only invited professional developers with a large experience with Java programming, from a convenient population

[3]http://refactoring.encs.concordia.ca/lambda-study/

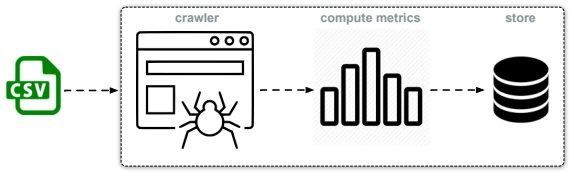


Figure 1: Overview of our automated approach for collecting code snippets and calculating metrics

of developers in our professional network. That is, professional developers correspond to our population of interest. The survey was organized in two sections, one for characterizing the experience of the participants and one for evaluating the benefits (or drawbacks) of introducing lambda expressions into legacy code. This second section comprises the following (survey) questions.

(SQ1) *Do you agree that the adoption of lambda expressions on the right code snippet improves the readability of the left code snippet?* This is a *likert scale* question—(1) meaning strongly disagree and (5) meaning strongly agree-that focus on the readability aspect.

(SQ2) *Which code do you prefer?* This is a *yes or no* question, which aims to understand if the new code increases general quality attributes. The same question has been explored in a previous work [7].

(SQ3) *Would you like to include any additional comment to your answers?* This is an open question that allow the participants to optionally present further details about their answers.

In the second phase, we replicate the survey, though only inviting undergraduate students. This decision allows us to understand the perceptions of novices regarding the introduction of lambda expressions into legacy Java code.

In both phases of the survey, the participants were randomly assigned to two distinct groups. In each group, the participants should answer the survey's questions for a set of a minimum three and a maximum of six pairs of code snippets—randomly selected from our population of 66 code snippets. Using this design, we could evaluate nine pairs of code snippets in the first survey; and 12 pairs of code snippet in the second—without demanding too much cognitive effort from the participants. To enforce this design, we dynamically generate a version of the survey for each participant, depending on the group and phase of the survey. As stated before, the participants also filled in some personal data (including gender, academic background, professional experience, and their experience with functional programming). Table 1 summarizes the characteristics of the participants of the first phase.

Initially, a pilot with a small number of students, five in total, was conducted to evaluate whether or not our tool settings were able to capture the opinion of the developers. After conducting this first pilot, several adjustments were made in the layout and in the functionality of the tools. After conducting the surveys, we cross-validate the results of the qualitative assessment with the results of the quantitative assessments, by correlating the results of the estimates for program comprehension from the two models discussed in the previous section with the results of the surveys. We also try to explain the results of the survey considering the

measurements of SLOC and CC, for all pairs of code snippets in the survey. Finally, we compare the results of the first and second phases of the survey using meta-analysis.

4 RESULTS

In this section we present the results of our studies. First we discuss the results of the quantitative assessment, which considers the models of Buse and Weimer [4] and Posnett et al. [18] (Section 4.1). After that, we present the results of the qualitative assessments and compare the findings of the two studies (Section 4.2). Finally, in Section 5 we present a general discussion that consolidates our understanding about the results of our work.

4.1 Quantitative Assessment

We considered the 66 pairs of selected code snippets during the quantitative assessment. For each pair, we calculated the number of lines of code (SLOC), the cyclomatic complexity (CC), the *estimate comprehensibility* using the Buse and Weimer and the Posnett et al. models. We address two main hypothesis in order to answer our research questions.

> **Hypothesis 1.** The introduction of lambda expressions improves program comprehension, according to the state-of-the-art readability models.

We use a signal test (Wilcoxon Signed-Rank Test [26]) to investigate this hypothesis, considering the comprehensibility assessments using the models of Buse and Weimer and Posnett et al. For each pair of code, the introduction of lambda expressions might have **increased**, **decreased**, or **unchanged** the comprehensibility, according to both models. Table 2 summarizes the results, considering all pairs of code snippets. The Wilcoxon Signed-Rank Test tests the *null hypothesis* that the comprehensibility of the source code *before* and *after* the introduction of lambda expressions are identical [26].

Surprisingly, although the Posnett et al. method builds upon the model of Buse and Weimer, we found conflicting results. The outcomes of the test reveals that the introduction of lambda expressions actually **decreases** program comprehension ($p - value < 0.0001$), when considering the Buse and Weimer model. Nonetheless, when we consider the Posnett et al. model, we cannot reject the null hypothesis, that is, the introduction of lambda expressions does not improve the comprehension of the code snippets ($p-value = 0.668$). Due to these conflicting results, we compare both models to the results of the qualitative assessment (Section 4.2).

> **Hypothesis 2.** SLOC and CC can be used to predict the benefits (or drawbacks) on program comprehension, according to the legibility models considered in this research.

We investigate this hypothesis using a regression model. First, we calculate the differences in the SLOC (Δs) and CC (Δcc) metrics, considering the code snippets before and after the introduction of lambda expressions. We then build two regression models, one considering as response variable the difference in the Buse and Weimer model (Δbw) and one considering as response variable the difference in the Posnett et al. model (Δp).

$$\Delta bw = b_0 + b_1\,\Delta s + b_2\,\Delta cc \quad (5)$$

$$\Delta p = c_0 + c_1\,\Delta s + c_2\,\Delta cc \quad (6)$$

Table 3 and Table 4 show the results of the regression analysis, considering the first and second models of Eq. (5) and Eq. (6). Considering a significance level < 0.05, we cannot predict the benefits / drawbacks of introducing lambda expressions, according to the Buse and Weimer and Posnett et al. models for estimating readability, in terms of lines of code and cyclomatic complexity. Therefore, we can refute our second hypothesis: it is not possible to estimate the effect on the readability metrics using SLOC and CC.

4.2 Qualitative Assessment

Considering the qualitative assessment, 28 participants (most of them with large experience in Java programming) evaluate a number between three and six pairs of code snippets. For each pair of code snippet, these participants answered the survey questions SQ1, SQ2, and SQ3. Recall that we split the code snippets into two groups, and thus each code snippet was evaluated by 14 participants. The data collection last 16 days, and, on average, each participant spent 2:30 minutes to evaluate each pair of code snippet.

We use two forms of data analysis in this assessments. First, we summarize the responses to SQ1 and SQ2 using tables and plots, to build a broad view of the answers for the closed questions. In the second analysis, we consider the answers to the open question, to draw qualitative findings. We present some of the tables, plots, and answers to the closed questions in the remaining of this section.

4.2.1 Improvements on Readability. The goal of the first question of our survey (*Do you agree that the adoption of lambda expressions on the right code snippet improves the readability of the left code snippet?*) is to evaluate if, according to the perception of Java developers, the introduction of lambda expressions improve the comprehension of the code snippets. We use a Likert scale to investigate this. Considering the answers to all pairs of code snippet, 39.7% and 11.1% either agree or strongly agree that the introduction of lambda expressions improve the readability of the code, respectively; while 24.6% of the responses are neutral, 21.4% disagree, and 3.2% strongly disagree with the SQ1 statement (see Table 5). Therefore, we found developers' leanings towards a readability improvement after the introduction of lambda expressions.

To better understand this result, we analyzed the answers for each pair of code snippet (see Figure 2). Transformations 1035, 1052, an 1180 present more than 60% of positive answers (i.e., introducing lambda expressions improves the readability of these code snippets). Differently, the pair of code snippet 1182 on Figure 3 received 79% of answers neutral or negative (i.e., the introduction of lambda expressions seems to reduce the readability of this code snippet). In this particular case, a `for(obj: collection) {...}` statement is replaced by a `collection.forEach(obj -> {...})` loop, which includes a lambda expression. Most of the participants did not agree that the introduction of a lambda expression improved the readability of the source code in this situation. One of the participants stated:

Table 1: Characterization of the Survey's Participants

ID	Gender	Degree	Experience Lambda	Experience functional programming	Experience
1	Male	Master Student	No	1-4 years	4 years
2	Male	BSc degree	Yes	1-4 years	2 years
3	Male	Master Student	Yes	More than five years	11 years
4	Male	BSc degree	Yes	1-4 years	4 years
5	male	Master Student	Yes	1-4 years	10 years
6	male	BSc degree	No	5+ years	11 years
7	male	Master Student	Yes	1-4 years	11 years
8	male	Master Student	Yes	More than five years	11 years
9	male	Master Student	No	No Experience	7 years
10	male	BSc degree	Yes	1-4 years	5 years
11	male	BSc degree	Yes	5+ years	5 years
12	male	PhD degree	Yes	No Experience	10 years
13	male	BSc degree	Yes	1 year	11 years
14	female	Master Student	No	No Experience	5 years
15	male	Master Student	Yes	No Experience	7 years
16	female	PhD degree	No	4-5 years	5 years
17	male	Master Student	Yes	1 year	4 years
18	male	BSc degree	Yes	1-4 years	2 years
19	female	Undergraduate Student	No	1 year	1 years
20	male	BSc degree	Yes	No Experience	7 years
21	male	Master Student	Yes	More than five years	11 years
22	male	Undergraduate Student	Yes	No Experience	1 year
23	male	BSc degree	Yes	1 year	1 year
24	male	Undergraduate Student	Yes	No Experience	1 year
25	male	Undergraduate Student	Yes	1 year	4 years
26	male	Master Student	Yes	4-5 years	5 years
27	male	BSc degree	No	No Experience	1 year
28	male	BSc degree	Yes	No Experience	11 years

Table 2: Number of pairs of code snippets that have improved the legibility, decreased the legibility, and unchanged the legibility; after the introduction of lambda expressions.

Model	Increased	Decreased	Unchanged
Buse and Weimer	13	44	9
Posnett et al.	31	35	0

Table 3: Summary o the regression model to estimate the difference on the Buse and Weimer estimates, using SLOC and CC

	Estimate	Std. Error	t value	Pr(>|t|)
(Intercept)	0.0309	0.0128	2.41	0.0190
Δs	0.0052	0.0029	1.77	0.0816
Δcc	0.0003	0.0199	0.01	0.9888

Table 4: Summary o the regression model to estimate the difference on the Posnett et al. estimates, using SLOC and CC

	Estimate	Std. Error	t value	Pr(>|t|)
(Intercept)	-0.0184	0.0161	-1.14	0.2567
Δs	-0.0088	0.0037	-2.41	0.0190
Δcc	0.0099	0.0249	0.40	0.6937

> "*(considering the code snippet 1182) I think that replacing a normal for each by an collection.forEach() would only bring benefits when there are additional calls either to the* `map` *or* `filter` *methods, or perhaps calls to some other method list processing.*"

Figure 4 shows the pair of code snippet 1180. In this example, an instance attribute (`duplicate`) is first initialized using an anonymous inner class (Figure 4-(a)). This anonymous inner class was later replaced by a lambda expression (Figure 4-(b)), and 64% of the participants either agree or strongly agree that this transformation

Figure 2: Answers to the first question of the survey, considering the pairs of code snippets

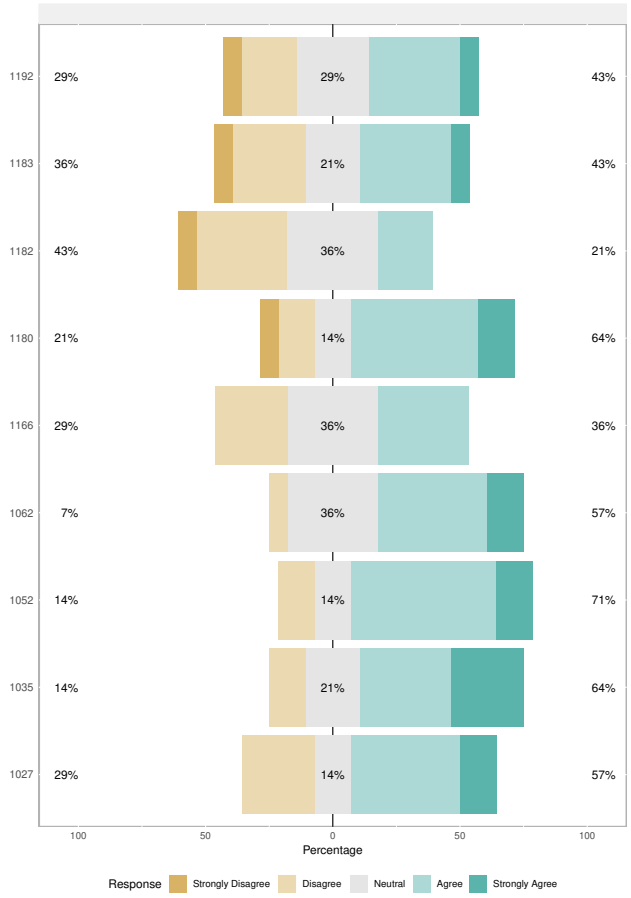


Table 5: Summary of the answers for the question *Do you agree that the adoption of lambda expressions on the right code snippet improves the readability of the left code snippet?*

SQ1	Answers	Percentage	Cum. Percentage
Strongly disagree	4	3.2%	3.2%
Disagree	27	21.4%	24.6%
Neutral	31	24.6%	49.2%
Agree	50	39.7%	88.9%
Strongly Agree	14	11.1%	100.0%
Total	126	100.0%	

improves the readability of the code snippet. Regarding this pair of code snippet, one of the participants state that:

> "*Here the transformation makes sense, because it eliminates the use of anonymous inner class with a trivial method body (often used to implement the Command design pattern in Java)*"

Considering all pairs of code snippets we use in the survey, only in two pairs of code snippets (1166 and 1182) we observe a

Figure 3: Pair of code snippet 1182

```
assertEquals(numRequests, responses.size());
for(TestResponse t: responses) {
  Response r = t.getResponse();
  assertEquals(t.method, r.getRequestLine().getMethod());
  ...
}
```

(a)

```
assertEquals(numRequests,responses.size());
responses.forEach(t -> {
  Response r = t.getResponse();
  assertEquals(t.method, r.getRequestLine().getMethod());
  ...
});
```

(b)

Figure 4: Pair of code snippet 1180

```
private Function duplicate = new Function() {
  publicString  apply(String  in) {
    returnin + in;
  }
};
```

(a)

```
private Function duplicate = (String  in) -> { return in + in; };
```

(b)

tendency towards either a neutral or a divergent opinion that the introduction of lambda expressions improves the readability of the code. More specifically, in these two cases, the percentage of agree and strongly agree was under 50%. Interesting, both are examples of transformations that replace a regular *for each* statement by a `collection.forEach(...)` using a lambda expression.

4.2.2 Source Code Preference. The goal of the second question of our survey (*Which code do you prefer?*) is to understand if the practitioners have a preference for the code before or after the introduction of lambda expression. Considering the nine pairs of code snippets of the survey (that we randomly select from the initial population), only the pair of code snippet 1166 received more choices for the first version of the code (i.e., before the introduction of lambda expressions). Therefore, we found some evidence in this survey that the participants identify the introduction of lambda expressions as a transformation that improves the quality of the source code. Surely, this preference depends on the experience of the developers, as one of the participants state:

> "*It depends on the practical knowledge on functional programming, since programmers of the 1980s and 1990s are likely to consider easier to understand code where loops, control variables, and pointers are explicit.*"

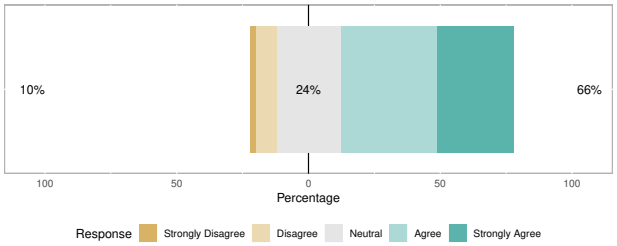
We used the Spearman correlation test to verify whether the reduction on lines of code and the reduction on cyclomatic complexity could explain the preference of the participants for the pieces of code after the introduction of lambda expressions. We found a moderate to high correlation (0.67) between the reduction on the

lines of code and the number of votes in favor of the code after the introduction of lambda expressions. Therefore, in the cases that a source code transformation to introduce lambda expressions reduce the number of lines of code, it might improve the general quality of the code—according to the perceptions of the participants. Differently, we found a weak correlation between the reduction on cyclomatic complex and the number of choice in favor (or against) of the code snippets using lambda expressions. This might be explained because the introduction of lambda expressions did not reduce the cyclomatic complexity in several cases.

4.2.3 Replication of the Qualitative Assessment. We replicate the qualitative assessment with Computer Science and Computer Engineering undergraduate students from our university, in order to understand the impact of introducing lambda expressions on code comprehension (that is, we focus only in the survey question SQ1). We follow the same procedures of the first survey. In particular, we randomly organized 12 pairs of code snippets in two different groups, and each student evaluated six pairs of code snippets. A total of fifteen students contributed with complete answers to the survey.

Figure 5 summarizes the results of the survey, considering all pairs of code snippets. In this study, 28.9% and 36.7% of the student agree or strongly agree, respectively, that the introduction of lambda expression in the code snippets improve readability. Only 7.8% of the answers disagree with the SQ1 statement, while 2.2% strongly disagree and 24% are neutral. We conducted a meta-analysis to understand how the perceptions of practitioners differ from the perceptions of the students. Meta-analysis is a quantitative procedure used to increase the confidence, combining the results of different empirical studies [5].

Figure 5: Summary of the students' answers to the first question of the survey



We conducted a meta-analysis to combine the results we collected from the answers of the surveys (with practitioners and students), considering only the first survey research question (SQ1). To this end, we computed the effect size and evaluated the summary effect and heterogeneity [3]. Since here we only combine the results of two studies, we use the model of fixed effects. Figure 6 shows the results using a forest plot—the most used method to present the results of a meta-analysis [3, 5]. A forest plot summarizes all effect data and the contribution of each study [3]. The first two rows of the plot represent the effect of the results of the two surveys (for the first survey question). The third and fourth rows of the plot highlight the comparison of the results, considering the fixed and random effect models. The first columns of the plot contains information about the studies (a brief description). It is possible to see the heterogeneity of the studies by interpreting the *odds ratio* information. The last columns of the plot show the confidence interval (considering 95% as reference) and the weights for the fixed and random models. Finally, the diamond on the last line of the forest plot reveals the overall size of the summary effect. The center of the diamond represents the size of the effect and its width represents the limits, considering a confidence interval of 95%.

According to the results, the effect size is significant. The values of $p = 0.65$ and $I^2 = 0\%$ indicates that the studies do not have heterogeneity. Since the meta-analysis measurement is on the right of the vertical line positioned in 1, there is a significant statistical effect. Both studies show an effect that argues in favor of the introduction of lambda expressions, in order to improve the readability of the source code. Most important, the lack of heterogeneity suggests that this result characterizes not only the perceptions of professionals with large experience in Java, but also the perceptions of undergraduate students with small exposure to functional programming.

5 DISCUSSION

As explained in the previous section, we found conflicting results in our research. First, the models for estimating readability diverge from one another. According to The Buse and Weimer [4] model, when we introduce a lambda expression into a Java legacy method, the readability of the method **decreases**. We found this using the non-parametric Wilcoxon Signed-Rank Test [26]. Differently, using the same test, the model of Posnett et al. [18] suggests that the introduction of lambda expressions does not impact program comprehension.

However, the results of the qualitative assessments, considering both developers with large experience in Java programming and undergraduate students, suggest that the introduction of lambda expressions improves program comprehension. We believe that these conflicting results are due to the limitations of both models on identifying improvements on code readability caused by finer-grained transformations. As a future work, we want to explore this issue using a catalog of well-known refactorings.

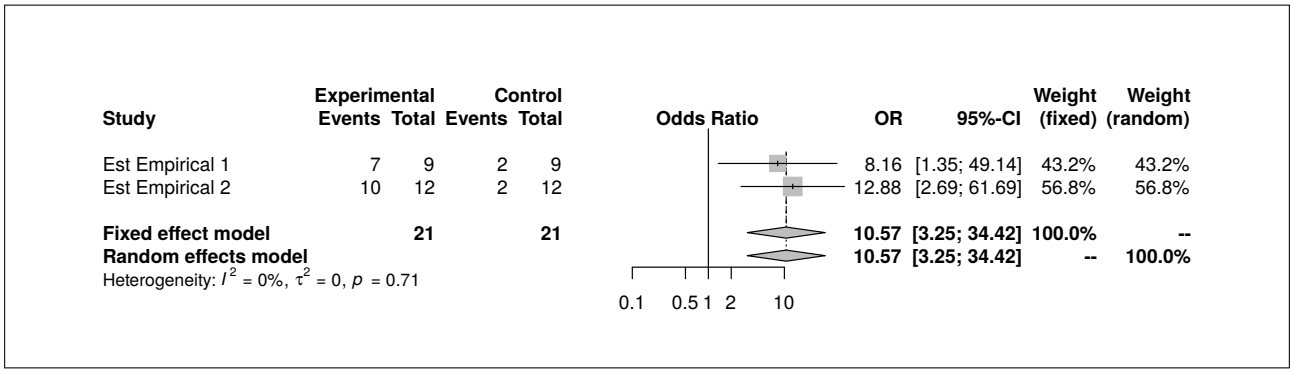
Considering the results of both quantitative and qualitative studies, we answer our research questions in Section 5.1 and present some lessons learned in Section 5.2. Finally, we present some threats to the validity of our study in Section 5.3.

5.1 Answers to The Research Questions

When using a mixed-method study, the best scenario occurs in situations where the results of each study supports and explains the results of one another. This is not the case here, and we are in favor of the results of the qualitative study, in particular due to some limitations of the existing models for estimating program comprehension. Therefore, considering our first research question (Does the use of lambda expressions improve program comprehension?), our findings reveal that refactoring legacy code introducing lambda expression improves program comprehension.

Regarding the second research question (*Does the introduction of lambda expressions reduce source code complexity?*), we found

Figure 6: Forest plot with the results of the meta-analysis



that considering all 66 pairs of code snippets, there is no evidence that introducing lambda expressions reduces the complexity of the code, considering SLOC and cyclomatic complexity. Considering our third research question (*What are the most suitable situations to refactor code to introduce lambda expressions?*), we found that replacing an anonymous inner class with simple method body is the most suitable situation to introduce lambda expression into legacy code. Composing different recursive patterns (e.g., filter, map, and collect) introduces another promising scenario for this type of refactoring. Differently, just replacing a simple *for over a collection statement* by a `collections.forEach()` does not bring any benefits, according to the participants of the survey.

Finally, regarding our fourth research question (*How do practitioners and students evaluate the effect of introducing a lambda expression into a legacy code?*), our findings reveal a positive leaning towards the adoption of lambda expressions, not only considering program readability (which we explore using SQ1), but also the preference of the code using lambda expressions (as we explore using SQ2).

5.2 Lessons Learned

Need for reviewing comprehensibility models. The state-of-the-art models for estimating code readability could not capture the benefits of introducing lambda expressions, as the participants of our survey report. We believe that a further investigation is necessary, in order to understand if these models fail to capture the benefits of fine-grained transformations similar to the introduction of lambda expression, or if they fail to general transformations such as popular refactorings. Nonetheless, both models are sensitive for code formatting decisions, including the number of blanking characters. Similar conclusions have been reported in a recent research work [8].

Recommendations for Refactoring Tools. We found that transforming anonymous inner classes with **simple method bodies** into lambda expressions is the scenario that brings more benefits for code comprehension. In this way, refactoring tools should not naively try to apply this transformation in the case that the method has several lines of code. Other scenarios of transforming legacy code into lambda expressions occur when it is possible to compose different recursive patterns, such as *filter* and *map*. Refactoring tools should also focus on these scenarios. Nonetheless, we consider that it is not recommended to apply automatic transformations of simple `for` statements over a collection into a `collections.forEach()` statement.

Use of Students on Program Comprehension Studies. The comparison of the surveys' results allowed us to conclude that it is worth to consider the opinion of students in program comprehension research, and the findings do not differ significantly when we compare their opinion with the opinion of professionals with large experience on software development. In this way we believe that we can generalize our results to both populations.

5.3 Threats to Validity

There are two main threats to our work. First, our results depend on the representativeness of the code snippets used in the investigation. Although we use a sample from real scenarios that introduce lambda expressions in legacy code, this sample might not correspond to representative population that would be recommended to draw conclusions from our quantitative assessment. Based on this first selection, and to try to avoid human bias, we also made a second random selection of code snippets to collect the opinion of the participants of the qualitative assessment. In the end, we evaluated nine pairs of code snippets in the first survey. During the replication study, in the second survey we considered the same pairs of the first survey, with three additional pairs of code snippets. This number is similar to the number of code snippets evaluated in a previous study [7].

The second threat is related to external validity. The surveys' participants belong to a relatively narrow group of professional developers with large Java experience and undergraduate students. Although they belong to a group of great interest to the software community, it was not possible to get a sample of participants large enough to generalize our findings. Nonetheless, since we combined the results of two surveys, we believe that we have a broad understanding about the benefits of introducing lambda expressions into Java legacy code. Finally, we could also have used other models to estimate readability, which have been previously discussed in the literature [21]. However, we only found an implementation of one of these models [4]. We still implemented the computation

for an additional model [18], but it would be difficult to provide implementations for all models available in the literature.

6 FINAL REMARKS

In this paper we presented the results of a mixed-method investigation (i.e., using quantitative and qualitative methods) about the impact on code comprehension with the adoption of lambda expressions in legacy Java systems. We used two state-of-the-art models for estimating code comprehension [4, 18], and found conflicting results. Although the model of Posnett et al. [18] reveals that the introduction of lambda expressions did not change the readability of the code, the model of Buse and Weimer [4] suggests that the introduction of lambda expressions actually decreases the comprehensibility of the source code. We also conducted two surveys with professional software developers and students, and the results of the surveys indicate that the introduction of lambda expressions in legacy code improves code comprehension. After considering these conflicting results, we argue that (a) this kind of source code transformation improves software readability for specific scenarios (e.g, replacing an anonymous inner class by a simple method body by a lambda expression) and (b) we need more advanced models to understand the benefits on program comprehension after applying finer-grained program transformations.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their valuable comments, which helped us to improve the quality of this paper. This work was partially supported by FAP-DF, research grant 05/2018.

REFERENCES

[1] Anwar Alqaimi, Patanamon Thongtanunam, and Christoph Treude. 2019. Automatically Generating Documentation for Lambda Expressions in Java. In *Proceedings of the 16th International Conference on Mining Software Repositories (MSR '19)*. IEEE Press, Piscataway, NJ, USA, 310–320. https://doi.org/10.1109/MSR.2019.00057

[2] Robert Baggen, José Pedro Correia, Katrin Schill, and Joost Visser. 2012. Standardized code quality benchmarking for improving software maintainability. *Software Quality Journal* 20, 2 (01 Jun 2012), 287–307.

[3] M. Borenstein, L.V. Hedges, J.P.T. Higgins, and H.R. Rothstein. 2011. *Introduction to Meta-Analysis*. Wiley. https://books.google.de/books?id=JQg9jdrq26wC

[4] Raymond P. L. Buse and Westley Weimer. 2010. Automatically documenting program changes. In *ASE 2010, 25th IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium, September 20-24, 2010*, Charles Pecheur, Jamie Andrews, and Elisabetta Di Nitto (Eds.). ACM, 33–42.

[5] H. Cooper, L.V. Hedges, and J.C. Valentine. 2009. *The Handbook of Research Synthesis and Meta-Analysis*. Russell Sage Foundation. https://books.google.de/books?id=LUGd6B9eyc4C

[6] Reno Dantas, Antonio Carvalho, Diego Marcilio, Luisa Fantin, Uriel Silva, Walter Lucas, and Rodrigo Bonifácio. 2018. Reconciling the past and the present: An empirical study on the application of source code transformations to automatically rejuvenate Java programs. In *25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018*, Rocco Oliveto, Massimiliano Di Penta, and David C. Shepherd (Eds.). IEEE Computer Society, 497–501.

[7] Rodrigo Magalhães dos Santos and Marco Aurélio Gerosa. 2018. Impacts of coding practices on readability. In *Proceedings of the 26th Conference on Program Comprehension, ICPC 2018, Gothenburg, Sweden, May 27-28, 2018*, Foutse Khomh, Chanchal K. Roy, and Janet Siegmund (Eds.). ACM, 277–285.

[8] Sarah Fakhoury, Devjeet Roy, Sk. Adnan Hassan, and Venera Arnaoudova. 2019. Improving Source Code Readability: Theory and Practice. In *Proceedings of the 27th International Conference on Program Comprehension (ICPC '19)*. IEEE Press, Piscataway, NJ, USA, 2–12. https://doi.org/10.1109/ICPC.2019.00014

[9] Jean-Marie Favre, Ralf Lämmel, Thomas Schmorleiz, and Andrei Varanovich. 2012. 101companies: A Community Project on Software Technologies and Software Languages. In *Objects, Models, Components, Patterns*, Carlo A. Furia and Sebastian Nanz (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 58–74.

[10] Dan Gopstein, Jake Iannacone, Yu Yan, Lois DeLong, Yanyan Zhuang, Martin K.-C. Yeh, and Justin Cappos. 2017. Understanding Misunderstandings in Source Code. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. ACM, New York, NY, USA, 129–139. https://doi.org/10.1145/3106237.3106264

[11] Alex Gyori, Lyle Franklin, Danny Dig, and Jan Lahoda. 2013. Crossing the Gap from Imperative to Functional Programming Through Refactoring. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013)*. ACM, New York, NY, USA, 543–553. https://doi.org/10.1145/2491411.2491461

[12] Raffi Khatchadourian, Yiming Tang, Mehdi Bagherzadeh, and Syed Ahmed. 2019. Safe Automated Refactoring for Intelligent Parallelization of Java 8 Streams. In *Proceedings of the 41st International Conference on Software Engineering (ICSE '19)*. IEEE Press, Piscataway, NJ, USA, 619–630. https://doi.org/10.1109/ICSE.2019.00072

[13] Davy Landman, Alexander Serebrenik, Eric Bouwers, and Jurgen J. Vinju. 2016. Empirical analysis of the relationship between CC and SLOC in a large corpus of Java methods and C functions. *Journal of Software: Evolution and Process* 28, 7 (2016), 589–618.

[14] Davood Mazinanian, Ameya Ketkar, Nikolaos Tsantalis, and Danny Dig. 2017. Understanding the Use of Lambda Expressions in Java. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 85 (Oct. 2017), 31 pages. https://doi.org/10.1145/3133909

[15] Jeffrey L. Overbey and Ralph E. Johnson. 2009. Regrowing a Language: Refactoring Tools Allow Programming Languages to Evolve. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '09)*. ACM, New York, NY, USA, 493–502. https://doi.org/10.1145/1640089.1640127

[16] Chris Parnin, Christian Bird, and Emerson R. Murphy-Hill. 2011. Java generics adoption: how new features are introduced, championed, or ignored. In *Proceedings of the 8th International Working Conference on Mining Software Repositories, MSR 2011 (Co-located with ICSE), Waikiki, Honolulu, HI, USA, May 21-28, 2011, Proceedings*, Arie van Deursen, Tao Xie, and Thomas Zimmermann (Eds.). ACM, 3–12. https://doi.org/10.1145/1985441.1985446

[17] Nancy Pennington. 1987. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology* 19, 3 (1987), 295 – 341. https://doi.org/10.1016/0010-0285(87)90007-7

[18] Daryl Posnett, Abram Hindle, and Premkumar T. Devanbu. 2011. A simpler model of software readability. In *Proceedings of the 8th International Working Conference on Mining Software Repositories, MSR 2011 (Co-located with ICSE), Waikiki, Honolulu, HI, USA, May 21-28, 2011, Proceedings*, Arie van Deursen, Tao Xie, and Thomas Zimmermann (Eds.). ACM, 73–82.

[19] Václav Rajlich. 2014. Software Evolution and Maintenance. In *Proceedings of the on Future of Software Engineering (FOSE 2014)*. ACM, New York, NY, USA, 133–144. https://doi.org/10.1145/2593882.2593893

[20] M. Riaz, E. Mendes, and E. Tempero. 2009. A systematic review of software maintainability prediction and metrics. In *2009 3rd International Symposium on Empirical Software Engineering and Measurement*. 367–377. https://doi.org/10.1109/ESEM.2009.5314233

[21] S. Scalabrino, M. Linares-Vásquez, D. Poshyvanyk, and R. Oliveto. 2016. Improving code readability models with textual features. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*. 1–10. https://doi.org/10.1109/ICPC.2016.7503707

[22] Margaret-Anne D. Storey, Kenny Wong, and Hausi A. Müller. 2000. How do program understanding tools affect how programmers understand programs? *Sci. Comput. Program.* 36, 2-3 (2000), 183–207.

[23] S. R. Tilley, S. Paul, and D. B. Smith. 1996. Towards a framework for program understanding. In *WPC '96. 4th Workshop on Program Comprehension*. 19–28. https://doi.org/10.1109/WPC.1996.501117

[24] N. Tsantalis, D. Mazinanian, and S. Rostami. 2017. Clone Refactoring with Lambda Expressions. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. 60–70. https://doi.org/10.1109/ICSE.2017.14

[25] Anneliese von Mayrhauser and A. Marie Vans. 1995. Program Comprehension During Software Maintenance and Evolution. *IEEE Computer* 28, 8 (1995), 44–55.

[26] Frank Wilcoxon. 1945. Individual Comparisons by Ranking Methods. *Biometrics Bulletin (JSTOR)* 1, 6 (1945), 80–83.