Journal of Software Engineering Research and Development, 2020, 8:7, doi: 10.5753/jserd.2020.744

Understanding the Impact of Introducing Lambda Expressions in Java Programs

Walter Lucas **University of Brasília (UnB)** *waltimlmm@gmail.com*
José Fortes **University of Brasília (UnB)** *jose.fortes.neto@gmail.com*
Francisco Vitor Lopes **University of Brasília (UnB)** *fvitorlopes@gmail.com*
Diego Marcílio **Università della Svizzera italiana** *dvmarcilio@gmail.com*
Rodrigo Bonifácio **University of Brasília (UnB)** *rbonifacio@unb.br*
Edna Dias Canedo **University of Brasília (UnB)** *ednacanedo@unb.br*
Fernanda Lima **University of Brasília (UnB)** *ferlima@unb.br*
João Saraiva **University of Minho** *saraiva@di.uminho.pt*

Abstract

Background: The Java programming language version eight introduced several 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 legacy code to introduce lambda expressions, besides other potential benefits, simplifies the code and improves program comprehension. **Aims**: The purpose of this work is to investigate this belief, conducting an in-depth study to evaluate the effect of introducing lambda expressions on program comprehension. **Method**: We conducted this research using a mixed-method approach. For the quantitative method, we quantitatively analyzed 158 pairs of code snippets extracted directly either from GitHub or from recommendations from three tools (RJTL, NetBeans, and IntelliJ). We also surveyed practitioners to collect their perceptions about the benefits on program comprehension when introducing lambda expressions. We asked practitioners to evaluate and rate sets of pairs of code snippets. **Results**: We found contradictory results in our research. Based on the quantitative assessment, we could not find evidence that the introduction of lambda expressions improves software readability—one of the components of program comprehension. Our results suggest that the transformations recommended by the aforementioned tools decrease program comprehension when assessed by two state-of-the-art models to estimate readability. Differently, our findings of the qualitative assessment suggest that the introduction of lambda expression improves program comprehension in three scenarios when: we convert anonymous inner classes to a lambda expression, use structural loops with inner conditional to an `anyMatch` operator, and apply structural loops to `filter` operator combined with a `collect` method. **Implications**: We argue in this paper that one can improve program comprehension when he/she applies particular transformations to introduce lambda expressions (e.g., replacing anonymous inner classes with lambda expressions). Also, the opinion of the participants highlights which kind of transformation for introducing lambda might be advantageous. This might support the implementation of effective tools for automatic program transformations.

Keywords: *Program Comprehension, Java Lambda Expressions, Empirical Studies*

1 Introduction

Software evolves to adapt to social and technical needs (Godfrey and German, 2008): users might request new features, or performance constraints must be met. Indeed, the success of a system depends on how easy its evolution is. If it does not change to reflect the needs of their users (Lehman and Ramil, 2001), it is doomed to failure. In the same vein, successful programming languages change over time (Overbey and Johnson, 2009): programmers require more features and more expressivity from language constructs.

Mainstream programming languages (e.g., Python and `C++`) also evolve to support new programming styles, such as the recent trend of imperative languages to adhere to the functional style. Since version 2.0, Python language supports features to facilitate list comprehension (Lott, 2018), a feature originally found in functional languages (like Erlang and Haskell). Similarly, `C++` introduced lambda expressions in `C++` version 11 (Stroustrup, 2013).

Recently, Java has adopted a faster release cycle to frequently deploy new features. Some of these releases did not significantly change the language semantics. Contrastingly, other releases present remarkable changes in language constructs. This is the case for Java 8, which introduces new features to facilitate functional programming and behavior parameterization. Using these features, developers can pass (anonymous) functions as arguments to other functions (Urma et al., 2014).

However, as languages evolve, programs' source code usually lag behind. When a language releases a new version, source code that was up-to-date suddenly becomes legacy code and older constructs often persist in the system while developers add new ones (Overbey and Johnson, 2009). The coexistence of old and new constructs puts a toll on programmers, requiring them to be familiar with different idioms that implement a similar behavior. To mitigate the problem of these old and new constructs coexisting, Overbey and Johnson (2009) recommended using refactoring tools that aim to help developers introduce new language constructs in legacy programs automatically.

For instance, Gyori et al. (2013) proposed a tool to rejuve-

nate Java programs that replaces legacy constructs, such as anonymous inner classes, with lambda expressions. The authors claim that the adoption of lambda expressions in Java improves program comprehension, though without presenting empirical evidence (Gyori et al., 2013). However, Dantas et al. (2018) report that this kind of transformation might not always improve the quality of the code, and developers often reject patches applying this kind of transformation (Dantas et al., 2018). Moreover, Mazinanian et al. (2017) found that developers often perform this kind of transformation without any tool support.

In previous work (Lucas et al., 2019), we investigated how the introduction of lambda expressions impacts source code comprehension. We found that state-of-the-art metrics to measure code readability fail to capture the benefits of introducing lambda expressions. Nonetheless, based on the findings of a survey with practitioners, we disclosed that the introduction of lambda expressions improve program comprehension only in a few specific scenarios, such using lambda expressions as a substitute to anonymous inner classes.

In this paper, we extend our previous work, mitigating two threats of that research: (a) the use of a small number of pairs of code snippets (each pair comprising the code before and after the introduction of a lambda expression) during the qualitative assessment; and (b) the use of real-world code snippets collected from open-source projects, whose versions after introducing lambda expressions could also have additional modifications (such as a bug fix). Therefore, we report the results of an extensive empirical investigation on the benefits of introducing lambda expressions in legacy code, considering 92 pairs of code snippets as suggested by automated tools. We review some aspects of our previous work and present new evidence about:

- **Scenarios that benefit from introduction lambda expressions**: We identified scenarios where the introduction of lambda expressions improve program comprehension. Tool developers might use this information to customize techniques that find opportunities to refactor a legacy code to use lambda expressions.
- **Lambda expressions make the code more succinct**: Our findings provide evidence that the introduction of lambda expressions makes the code more succinct (in more than 80% of the scenarios, the total number of lines of code reduced after introducing lambda expressions)—even though this does not necessarily lead to an improvement on code comprehension.
- **Lambda expressions make debugging difficult**: Our results suggest that the introduction of lambda expressions can lead to pieces of code that are harder to debug. We consider this as a possible negative side effect of introducing lambda expressions.
- **Relevance of tooling support for rejuvenating Java code**: We also found that developers consider tooling support to be important for performing transformations introducing lambda expressions in Java legacy code. Nonetheless, existing tools also recommend transformations that need manual improvements, lead to small benefits, or make the code harder to understand.

2 Background and Related Work

Program comprehension is a fundamental software attribute that facilitates its maintenance and supports its evolution (von Mayrhauser and Vans, 1995). Understanding existing software enables maintainers to successfully evolve functionality and/or integrate improvements for every type of change commonly associated with software maintenance and evolution, including adaptive, perfective, and corrective modifications (von Mayrhauser and Vans, 1995). Understanding software is challenging due to several factors, one of which is that large programs are often maintained by developers with different skills and using different practices (Storey et al., 2000). Moreover, in many cases, the source code may be the only available and up to date reference for a software (Storey et al., 2000), though poor design and lack of good programming practices might compromise program comprehension (Tilley et al., 1996).

The practices developers use to understand a software are diverse and often are task-related (e.g., documenting part of a system, fixing a bug, and implementing a new feature). Indeed, "programmers use domain knowledge, programming knowledge, and comprehension strategies when attempting to understand a program" (Tilley et al., 1996). Program comprehension uses existing knowledge to acquire new knowledge to build a mental model of the software that might help developers accomplish a specific task (von Mayrhauser and Vans, 1995). While it is true that the skills and experiences of a developer are relevant when he/she wants to understand 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 as defined by Pennington (1987). Conversely, the use of some *obscure programming constructs* (e.g., atoms of confusion) increases the rate of source code misunderstandings (Gopstein et al., 2017). For instance, the atoms of confusion *conditional operator* and *logical as control flow*[1] involve fundamental language constructs such as math operators and if statements.

Although many software characteristics might impact program comprehension (e.g., variable names (Avidan and Feitelson, 2017) and atoms of confusion (Gopstein et al., 2017)), in this paper, we are particularly interested in aspects related to source code quality that might either facilitate or hinder program understanding (Storey et al., 2000). Several research studies (Buse and Weimer, 2010; Posnett et al., 2011; Scalabrino et al., 2016) have explored the use of models for estimating the readability of the source code—which directly affect program comprehension. Additionally, previous research has already investigated the impact of coding practices on software readability (Gopstein et al., 2017; dos Santos and Gerosa, 2018). Our work builds upon these previous efforts, using existing models for estimating software readability (Buse and Weimer, 2010; Posnett et al., 2011), and procedures to qualitatively assess the preference of developers when considering sets of code snippets (dos Santos and Gerosa, 2018). We apply these models in a different and

[1] https://atomsofconfusion.com/

particular scenario: the introduction of lambda expressions into Java legacy code.

Lambda expressions were introduced in Java 8 to support functional programming (Tsantalis et al., 2017), lifting function definitions to values, thus allowing developers to pass a lambda function definition as an argument to a method (Alqaimi et al., 2019). Developers can also use lambda expressions in Java to abstract parallelism and remove the boilerplate code necessary to write anonymous inner classes (Alqaimi et al., 2019). Moreover, lambda expressions enable chaining functional recursive patterns (e.g., *map* and *filter*) using the `stream` API methods as an alternative way to iterate, filter, and collect data from a collection (Mazinanian et al., 2017). For instance, consider the code snippets in Figure 1 (**filter 1** and **filter 2**), based on an implementation of the *101Companies* problem domain (Favre et al., 2012). In this example, the goal is to filter a department's employees 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.

Figure 1. Filtering employees with high salaries, being **filter1** approach previous to Java 8 and **filter2** approach using lambda expressions and 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;
}
```

filter 1

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

filter 2

Previous research on Java lambda expressions focused on their introduction via automatic techniques for refactoring legacy code to "make the code more succinct and readable" (Gyori et al., 2013; Dantas et al., 2018)—in particular situations that one 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 (Tsantalis et al., 2017) and to use parallel features of Java 8 properly (Khatchadourian et al., 2019). Also, Mazinanian et al. (2017) present a comprehensive study on the adoption of Java lambda expressions to understand the motivations that lead Java developers to adopt the functional style of thinking in Java. The authors published a large dataset with more than 100 000 real usage scenarios. We use this dataset to understand program comprehension benefits with the adoption of Java lambda expressions.

At first glance, the use of lambda expressions, due to its conciseness, yields a more succinct and readable code (Gyori et al., 2013; Dantas et al., 2018). However, this is not always the case, as Dantas et al. (2018) produced automated refactorings for iterating on collections that developers judged less comprehensible. We aim to investigate further which scenarios benefit 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 a Java method to introduce a lambda expression, and thus answering the research questions we present in Section 3.1. To this end, we conducted a research in two phases, both using a mixed-methods approach.

In the first phase, whose results we presented in previous work (Lucas et al., 2019), we carried 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. We also conducted a qualitative investigation (survey) considering the opinion of 28 practitioners that answered questions that also aim to compare the code before and after the introduction of lambda expressions in nine pairs of code snippets.

In the second phase we mitigated some possible threats that we identified in the first study: a small number of code snippets used in the survey of the first phase and the assessment of code snippets that might contain not only a manual program transformation, but actually a manual program transformation and an additional contribution to the program (e.g., a bug fix). As such, in the second phase we leveraged existing support of program transformation tools to refactor legacy code of open source systems to introduce lambda expressions. Considering the outcomes of these program transformation tools, we again conducted a quantitative assessment (using state-of-the-art models for measuring software comprehension) of a random sample of 92 pairs of code snippets and a survey with 182 practitioners that evaluated at least five code snippets from this sample of 92 pairs.

3.1 Research Questions

We investigated the following research questions in our study.

(Q1) *Does the use of lambda expressions improve program comprehension?*

(Q2) *Does the introduction of lambda expression reduce source code complexity?*

(Q3) *What are the most suitable situations to refactor a code to introduce lambda expressions?*

(Q4) *How do practitioners evaluate the effect of introducing a lambda expression into a legacy code?*

(Q5) *What is the practitioners' opinion about the recommendations from automated tools to introduce lambda expressions?*

We conducted this research using an iterative approach, and after investigating a given question, new sub-questions and hypothesis emerged. For instance, we investigated whether or not the reduction in the size of a code snippet, after introducing a lambda expression, has an influence on the perception of the participants about the quality of the transformation.

3.2 Metrics of the Quantitative Study

We measured 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 (Riaz et al., 2009; Baggen et al., 2012; Landman et al., 2016). In addition, we used two models to estimate and compare the readability of each pair of code snippets considered in our research. Readability is one of the aspects used for assessing program comprehension, and hereafter both terms (readability and program comprehension) are used interchangeably. The first model we used to estimate program comprehension is based on the work of Buse and Weimer (2010). It estimates the comprehensibility of a code snippet considering a regression model that takes as input several characteristics, including the length of each line of code in a code snippet, the number of identifiers in a code snippet, and the length of the identifiers present in a code snippet (Buse and Weimer, 2010).

The second model was proposed by Posnett et al. (2011), which builds upon the Buse and Weimer model, though considering a smaller number of characteristics. Based on this model, we can estimate the readability of a code snippet using Eq. (1) and Eq. (2); and the constant $C = 8.87$.

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

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

That is, in the Posnett et al. model, we calculated 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 used an existing tool[2] to estimate the comprehensibility of the code snippets using the Buse and Weimer (2010) model. We developed our own tool to automate the computation of the comprehensibility model by Posnett et al. (2011).[3] We executed these computations for all pairs of code snippets that we collected either from real scenarios (first phase) or from the outcomes of the program transformation tools (second phase).

3.3 Code Snippets' Datasets

In the first phase of this research, we used an existing tool (MinerWebApp) and a dataset from a previous work (Mazinanian et al., 2017), to identify code snippet candidates to our research. MinerWebApp 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 (Mazinanian et al., 2017). The goal of MinerWebApp is to identify and classify the use of lambda expressions code snippets. MinerWebApp classifies the occurrences of lambda expressions into three categories:

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

The decision of using an existing tool and dataset simplified our process of collecting real usage scenarios of lambda expressions. We randomly selected 59 code snippets from the MinerWebApp dataset—considering exclusively the code snippets of the third category (*Existing method*). We also collected 29 code snippets of *refactoring scenarios* we generated using RJTL (Dantas et al., 2018) and submitted via pull requests to open source projects. In total, we selected 88 code snippets from 22 projects, including code snippets from the Elastic Search, Spring Framework, and Eclipse Foundation projects. We manually reviewed these code snippets and removed 22 pairs that clearly do not correspond to a refactoring or that already had a lambda expression in the first version of the code. This cleanup lead to a final dataset with 66 pairs of code snippets from 19 projects that we considered in the first phase of the research. In Table 1 we show the number of pairs of code snippets we collected from the GitHub repositories, coming either from MinerWebApp or from RJTL transformations.

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

[2] http://www.arrestedcomputing.com/readability/

[3] https://github.com/rbonifacio/program-comprehension-metrics

Table 1. Selected projects in the first phase.

Project	Snippets from MinerWebApp	Snippets from RJTL
seleniumQuery	0	10
elasticsearch	0	4
CoreNLP	0	15
vertx-examples	2	0
Swagger2Markup	2	0
SpongeAPI	4	0
tailor	2	0
Agrona	1	0
RxAndroidBle	2	0
optaplanner	7	0
RxJava-Android-Samples	3	0
kaa	1	0
jersey	4	0
uhabits	2	0
graylog2-server	1	0
FluentLenium	1	0
qualitymatters	1	0
jbpm	1	0
spring-integration	3	0

metrics (Figure 2 shows an overview of the approach). The 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, and the type of the refactoring (e.g., anonymous inner class to lambda expression, `foreach` statements to a recursive pattern using lambda expressions, and so on).

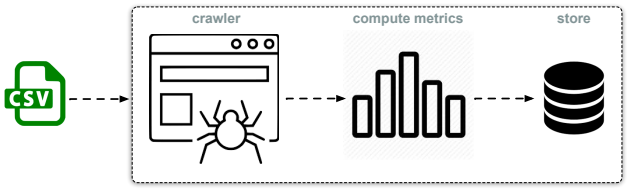


Figure 2. Procedures for collecting code snippets and calculating metrics

In the second phase, we used three automated refactoring tools (RJTL tool, NetBeans IDE, and Intellj IDE) to find opportunities and then introduced lambda expressions globally into the methods of five open-source systems (see Table 2). We chose these systems because they have been used to assess the performance of Lambdaficator (Gyori et al., 2013)—lately integrated into NetBeans to assist developers to migrate legacy systems towards Java 8. We were also able to build and execute the test cases of these systems, before and after applying the transformations. After executing the three tools in the five systems, we generated a dataset of 1987 transformations recommending refactorings to introduce lambda expressions (Table 2 shows the details).

Table 2. Number of refactoring recommendations each tool (RJTL, NetBeans IDE, and IntelliJ) produced.

Project	RJTL	NetBeans IDE	IntelliJ IDE
junit4-r4.13-rc-2	9	104	39
tomcat-7.0.98	3	354	105
fitnesse-20191110	4	319	70
antlrworks-1.5.1	89	316	118
ant-ivy-rel-2.5.0	23	389	45

We followed a set of steps in order to validate and create our second dataset of transformations. We first downloaded and built the last (stable) version of the systems, before executing the refactoring tools. After that, for each program transformation tool, we created a specific Git branch, executed the program transformation tool, and built the system again—looking either for a compilation or test execution failure. We checked out the files that, after applying a transformation, introduced a failure, removing spurious transformations. Accordingly, we built a dataset with 1987 transformations. We then randomly selected 92 pairs of code snippets to explore in the second phase of our research. We classified this final set of 92 transformations (Appendix A details the taxonomy) and computed the source code metrics and readability models. We stored the code snippets and the results of the metric calculations into a database. Table 3 summarizes this final set of 92 transformations.

Table 3. Number of transformation grouped by Type and Tool.

Type	RJTL	Netbeans IDE	IntelliJ IDE
Anonymous inner class	17	13	28
Reduce	0	2	0
Chaining	0	6	0
ForEach	0	9	0
Map	0	2	0
Filter	2	1	0
AnyMatch	12	0	0

We finally investigated the situations where at least two tools recommended a refactoring in the same code snippet. Considering the initial set of 1987 transformations, we found 357 cases (17.96%) of code snippets having recommendations from more than one tool. Nonetheless, the recommendations are not exactly the same. For instance, the code snippets of Figure 3 present transformations recommended to the same original code (Figure 3-(a)), but suggested by NetBeans IDE, IntelliJ IDE, and RJTL. In this example, it is possible to realize that the IntelliJ IDE leverages the mechanism of type inference, while NetBeans IDE and RJTL do not. Moreover, there is a slight difference in the indentation of the resulting code from the NetBeans IDE and RJTL recommendations. We removed this kind of duplication in our dataset with 100 code snippets, leading to a final dataset of 92 pairs of code that we used in the second phase of our research.

3.4 Procedures of the Qualitative Study

Regarding the qualitative study, we conducted the research using an approach based on a previous work (dos Santos and Gerosa, 2018). That is, we designed an online survey that allowed the participants to evaluate pairs of code snippets. In the first phase we only invited professional developers with some background in Java programming, from a convenient population of developers in our own professional network. Table 7 details the characteristics of the survey participants from the first phase of our research. The survey was organized in two sections. The first section aimed to characterize the experience of the participants; while the second one aimed to investigate the benefits (or drawbacks) of introduc-

Figure 3. Transformations recommended to the same code snippet suggested by RJTL, NetBeans,and IntelliJ.

```
public XJRotableToggleButton createToggleButton(String title...) {
  XJRotableToggleButton b = new XJRotableToggleButton(title);
  b.setFocusable(false);
  b.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      performToggleButtonAction(tag);
    }
  });
  components2toggle.put(c, b);
  return b;
}
```

(a) Original Code

```
public XJRotableToggleButton createToggleButton(String title...) {
  XJRotableToggleButton b = new XJRotableToggleButton(title);
  b.setFocusable(false);
  b.addActionListener((ActionEvent e) →{
    performToggleButtonAction(tag);
  });
  components2toggle.put(c, b);
  return b;
}
```

(b) Code after applying the NetBeans transformation

```
public XJRotableToggleButton createToggleButton(String title...) {
  XJRotableToggleButton b = new XJRotableToggleButton(title);
  b.setFocusable(false);
  b.addActionListener(e →performToggleButtonAction(tag));
  components2toggle.put(c, b);
  return b;
}
```

(c) Code after applying the IntelliJ transformation

```
public XJRotableToggleButton createToggleButton(String title...) {
  XJRotableToggleButton b = new XJRotableToggleButton(title);
  b.setFocusable(false);
  b.addActionListener((ActionEvent e)→ { performToggleButtonAction(tag); } );
  components2toggle.put(c, b);
  return b;
}
```

(d) Code after applying the RJTL transformation

ing lambda expressions into legacy code. This second section comprised the following (survey) questions.

- **S1Q1:** *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, which focuses on the readability aspect.
- **S1Q2:** *Which code do you prefer?* This is a yes or no question, which aims to understand if the new code improves general quality attributes. The same question has been explored in a previous work (dos Santos and Gerosa, 2018).
- **S1Q3:** *Would you like to include any additional comment to your answers?* This is an open question that allowed the participants to optionally present further details about their answers.

We first conducted a pilot with five students, to evaluate whether our online survey tool would be able to properly capture the opinion of the developers. After conducting this pilot, we implemented several adjustments in the layout and in the functionalities of the tool, in order to increase our confidence in the tool for the next executions of the *survey*. The pilot also revealed that answering all pairs of code snippets was a time-consuming activity. For this reason, we split the pairs of code snippets into two groups, and then randomly assigned the participants to answer the survey questions considering code snippets either from the first or from the second 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 the first or second groups of code snippets.

Considering the second phase of our study, we used the set of 92 randomly selected pairs of code snippets whose transformed code correspond to a recommendation from RJTL, NetBeans IDE, or IntelliJ. In this phase, the participants answered the following questions.

- **S2Q1:** *What is your opinion about the following sentences?*
  (a) The new code is easier to comprehend,
  (b) The new code is more succinct and readable,
  (c) The intention of using a lambda expression in the new code is clear, and
  (d) The new code is harder to debug.

  Respondents presented their opinion about these sentences using a *Likert scale*—(1) meaning Strongly disagree and (5) meaning Strongly agree. The first three sentences are claims that motivate the adoption of Lambda expressions in Java programs (Gyori et al., 2013). The fourth sentence came from our own experience in debugging pieces of code that use Java lambda expressions.
- **S2Q2:** *How often would you perform this type of transformation?* This is a *Likert scale* question— (1) meaning Never and (5) meaning Always. The goal was to evaluate how often developers would perform a specific transformation to introduce lambda expressions.
- **S2Q3:** *How important is the automated support for this kind of transformation?* This is a *Likert scale* question— (1) meaning Not important at all and (5) meaning Extremely important. The goal of this question was to evaluate how important the use of tools to support a specific transformation is.
- **S2Q4:** *Would you perform this transformation? Why?* This is an open question that allowed the participants to optionally present further details about their opinion.

In the second phase of our research, we used a set of social media tools to invite developers to answer the survey. That is, we sent a message to specific communities of Java Developers, including communities from Facebook, Reddit, Telegram, and mailing list of Java developers (e.g. NetBeans Developers, JDK Developers). We presumed that the developers have a good experience with Java programming. This phase had 182 participants located in 32 different countries (see Table 4). The developers needed 04:23 minutes (on average) to complete the questionnaire, where they evaluated a maximum of 5 transformations and answered a set of 7 questions regarding each pair of code snippet. In this phase, we generated a survey randomly selecting five pairs of code snippets for each participant. Tables 5 and 6 summarize the number of participants considering the level of education and professional experience of the respondents, respectively.

Table 4. Distribution of respondents according to their location.

Country	Respondents	Percentage
Brazil	71	39.01
United States	25	13.74
Germany	16	8.79
Portugal	8	4.40
India	7	3.85
United Kingdom	6	3.30
Netherlands	5	2.75
Spain	4	2.20
Other countries	40	21.97

Table 5. Characterization of the Survey's Participants in the second phase over the level of education.

Developers Degree	**Number of participants**	**Percentage**
Some high school	5	2,74%
High school graduate	13	7,14%
Undergraduate	21	11,53%
Bachelor's degree	58	31.86%
Master's degree	76	41.75%
Doctorate degree	9	4.94%

Table 6. Characterization of the Survey's Participants in the second phase over developer experience.

Developers Experience	**Number of participants**	**Percentage (%)**
Less than one year	14	7.69%
Between one and four years	52	28.57%
Between five and ten years	48	26.37%
More than ten years	68	37.36%

We cross-validated 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 explored the results of the survey considering the measurements of SLOC and CC, for all pairs of code snippets in the survey.

3.5 Data Analysis

We used exploratory data analysis (EDA) to answer our first two research questions. EDA is a method that allows researchers to build a broad understanding about the data, using descriptive statistics (e.g., median and mean) and graphical methods (e.g., histograms and boxplots). We also leveraged hypothesis testing to further explore the first two research questions.

Regarding the remaining research questions, which we addressed using surveys as the main method for data collection, we also relied on EDA to consolidate the answers to the Likert scale based questions (in terms of descriptive statistics and plots); while the answers to the survey's open-end questions were literally quoted. Since we collected a more significant feedback for the open-ended questions in the second phase of the research (177 answers in total), we also consolidated the answers to the second phase's open-ended question using Thematic Analysis (Silva et al., 2016; Shrestha et al., 2020).

We conducted our thematic analysis in four steps. In the first, we carried out an initial reading of the answers to the fourth question of our survey (S2Q4), preparing the scene before starting the coding stage. In the second step, we performed an initial coding for each answer. Next, in the third stage, we analyzed the codes with the goal of finding themes (that is, grouping of related codes). Finally, in the fourth step, we reviewed and merged the themes, generating a new, more comprehensive list of topics. We included a small phase of cross-validation, in which two authors gave feedback on the assignments. These two authors did not contribute to the initial assignment of codes and themes to the answers.

4 Results of the First Phase

In this section we present the results from the first phase of our research. Initially we discuss the outcomes of the quantitative assessment, which considers the models of Buse and Weimer (2010) and Posnett et al. (2011) (Section 4.1). After that, we present the results of the qualitative assessments and compare the findings of the two studies (Section 4.2).

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 addressed two main hypothesis in order to answer our research questions.

> **H1:** The introduction of lambda expressions improves program comprehension, according to the state-of-the-art readability models.

Conversely, our first null hypothesis ($H1_0$) investigates whether *the introduction of lambda expressions does not change program comprehension, according to state-of-the-art readability models*. We used a signal test (Wilcoxon Signed-Rank Test Wilcoxon (1945)) 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. As such, the Wilcoxon Signed-Rank Test tested the null hypothesis that the comprehensibility of the source code *before* and *after* the introduction of lambda expressions are identical (Wilcoxon, 1945). Table 8 summarizes the results, considering all pairs of code snippets.

Although the Posnett et al. method builds upon the model of Buse and Weimer, our analysis revealed a lack of agreement in the results from the two models. The outcomes of the test revealed that the introduction of lambda expressions actually **decreases** program comprehension (p-value < 0.0001), when considering the Buse and Weimer model. Nonetheless, when we considered the Posnett et al. model, we could not reject the null hypothesis, and this result suggested that the introduction of lambda expressions does not affect the comprehension of the code snippets (p-value $= 0.668$). Due to these conflicting results, we compared both

Table 7. Characterization of the Survey's Participants in first phase.

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 8. Number of pairs of code snippets that have increased the readability, decreased the readability, and unchanged the readability; after the introduction of lambda expressions.

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

models to the results of the qualitative assessment (Section 4.2).

> **H2.** SLOC and CC can be used to predict the benefits (or drawbacks) on program comprehension, according to the readability models considered in this research.

We investigated this hypothesis using a regression model. First, we calculated the differences in the SLOC (Δs) and CC (Δcc) metrics, considering the code snippets before and after the introduction of lambda expressions. We then built 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).

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

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

Accordingly, we unfolded **H2** in two alternative hypotheses, one for each readability model. That is, the null hypotheses for **H2** are as follows.

- $H2.1_0$: There is no relationship between Δbw and the predictors Δs and Δcc.
- $H2.2_0$: There is no relationship between Δp and the predictors Δs and Δcc.

Table 9 and Table 10 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 could not predict the benefits/drawbacks of introducing lambda expressions, according to the Buse and Weimer model to estimate readability, in terms of lines of code (p-value = 0.08) and cyclomatic complexity (p-value = 0.98). This result suggested that we should not reject the null hypothesis $H2.1_0$, and there is a negligible relationship between the predictors (Δs and Δcc) with the response variable Δbw. Finally, only 2% of the variability in Δbw was explained by the linear regression of Eq. (5) (Adjusted R-squared: 0.02). Similarly, variables Δs and Δcc did not explain the variability in Δp (Adjusted R-squared: 0.05). Nonetheless, considering the second regression model (Eq. (6)), the result suggested that there is a relationship between SLOC and Δp (p-value = 0.01)—though it is a small correlation ($\rho = -0.188$ using the Spearman correlation method).

In summary, the results of the regression analysis refuted our hypothesis **H2**: Δs and Δcc presented a negligible re-

lationship with Δbw and Δp; and thus they could not adequately predict the variability in the response variables of Eq. (5) and Eq. (6).

Table 9. Summary of 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 10. 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

4.2 Qualitative Assessment

Considering the qualitative assessment, 28 participants with a substantial experience in Java programming evaluated a number between three and six pairs of code snippets. For each pair of code snippet, these participants answered the survey questions S1Q1, S1Q2, and S1Q3. Recall that we split the code snippets into two groups, and thus each code snippet was evaluated by 14 participants. The data collection lasted 16 days, and, on average, each participant spent 2:30 minutes to evaluate each pair of code snippet.

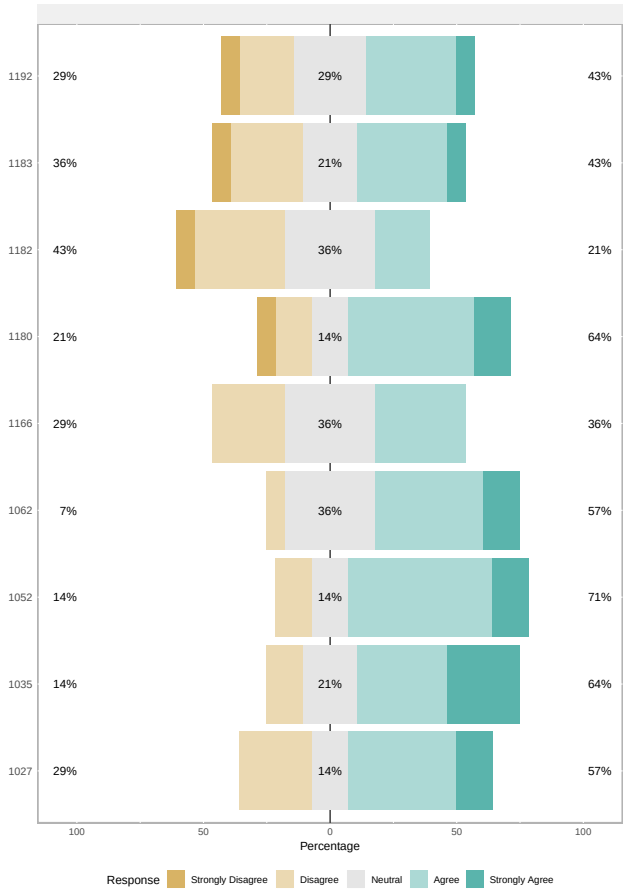
We used two forms of data analysis in this assessment. First, we summarized the responses to SQ1 and SQ2 using tables and plots, which allowed us to build a broad view of the closed questions' answers. In the second analysis, we considered the answers to the open questions literally (some of them are quoted here), to draw a broader understanding about the implications of refactoring Java legacy code to introduce lambda expressions.

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?*) was to evaluate if, according to the perception of Java developers, the introduction of lambda expressions improve the comprehension of the code snippets. We used a Likert scale to investigate this. Considering the answers to all pairs of code snippet, 11.1% and 39.7% either strongly agree or agree that the introduction of lambda expressions improve the readability of the code, respectively; while 24.6% of the responses were neutral, 21.4% disagree, and 3.2% strongly disagree with the SQ1 statement (see Table 11). Therefore, we found developers leaning 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 4). Transformations 1035, 1052, and 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 5 received 79% of answers either 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:

Figure 4. Answers to the first question of the survey, considering the pairs of code snippets



> "*(considering the code snippet 1182) I think that replacing a normal for each by a `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 6 shows the pair of code snippet 1180. In this example, an instance attribute (`duplicate`) was first initialized using an anonymous inner class (Figure 6-(a)). This anonymous inner class was later replaced by a lambda expression (Figure 6-(b)), and 64% of the participants either agree or strongly agree that this transformation improves the readability of the code snippet. Regarding this pair of code snippet, one of the participants stated that:

Table 11. 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?*

S1Q1	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%	

Figure 5. 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)

> "*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)*"

Figure 6. Pair of code snippet 1180

```
private Function duplicate = new Function() {
   public String apply(String in) {
      return in + in;
   }
};
```

(a)

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

(b)

Considering all pairs of code snippets we used in the survey, only in two pairs of code snippets (1166 and 1182) we observed a tendency towards either a neutral or a disagreement 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%. Both are examples of transformations that replace a regular *for each* statement to 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?*) was to understand if the practitioners had 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 selections 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 reduced the number of lines of code, it might have also improved the general quality of the code—according to the perceptions of the participants. Differently, we found a weak correlation between the reduction on cyclomatic complexity and the number of choices in favor (or against) of the code snippets using lambda expressions. We could understand this result because the introduction of lambda expressions did not reduce the cyclomatic complexity in several cases.

5 Results of the Second Phase

In this section, we replicate the process executed in the first phase, but only considering transformations suggested by automated tools. Section 5.1 presents the results of the quantitative assessment, taking into account the models of Buse and Weimer (2010) and Posnett et al. (2011). After that, we present the results of the qualitative assessments and compare them to the results of the quantitative study (Section 5.2).

5.1 Quantitative Assessment

We considered the 92 pairs of code snippets randomly selected from the set of recommendations to introduce lambda expressions suggested by RJTL, NetBeans, and IntelliJ. For each pair, we estimated the *code comprehension* of the versions before and after applying the suggested transformations, using both the Buse and Weimer (2010) and Posnett et al. (2011) models. We also calculated the SLOC and CC metrics for both versions of code snippets.

To investigate **H1** (*The introduction of lambda expressions improves program comprehension, according to state-of-the-art readability models*), we executed the Wilcoxon Signed-Rank Test considering the two models to measure code comprehension. First, we evaluated the situations where a transformation **increased**, **decreased** or **unchanged** code comprehension according to the models. After that, we executed

the *Wilcoxon Signed-Rank Test*. Table 12 summarizes the results, showing that, in most of the cases, the introduction of lambda expressions suggested by automated tools actually reduces code comprehension, according to both state-of-the-art readability models.

Table 12. Number of pairs of code snippets that have increased the readability, decreased the readability, and unchanged the readability; after the introduction of lambda expressions.

Model	Increased	Decreased	Unchanged
Buse and Weimer	25	63	4
Posnett et al.	20	67	5

The results of the Wilcoxon Signed-Rank Test suggested that the introduction of lambda expressions decreases the comprehensibility of the pairs of code snippets (p-value $<$ 0.0001). For instance, Figures 7 and 8 show pairs of snippets that have been evaluated using the readability metrics. The transformation of an anonymous inner class led to an improvement according to Buse and Weimer (2010) metric: the readability for the code before the transformation according to this model is 0.29; and 0.50 after introducing a lambda expression. However, considering a transformation that replaces a *for loop* by a lambda expression, the metric's result worsened significantly, reducing from 0.72 to 0.13 after the source code transformation.

Figure 7. Pair of code snippet 528. Replacing an anonymous inner class.

```
public synchronized String getResolverName(ModuleRevisionId mrid) {
 ModuleSettings ms = moduleSettings.getRule(mrid, new Filter<ModuleSettings>()
 {
   public boolean accept(ModuleSettings o) {
    return o.getResolverName() != null;
   }
 });
 return ms == null ? defaultResolverName : ms.getResolverName();
}
```

(a)

```
public synchronized String getResolverName(ModuleRevisionId mrid) {
 ModuleSettings ms = moduleSettings.getRule(mrid, (ModuleSettings o)→{
   return o.getResolverName() != null
 ;});
 return ms == null ? defaultResolverName : ms.getResolverName();
}
```

(b)

To investigate the **H2** hypothesis (*SLOC and CC can be used to predict the benefits (or drawbacks) on program comprehension, according to the readability models considered in this research.*), we calculated the differences in the SLOC (Δs) and CC (Δcc) metrics, considering the code snippets before and after the introduction of lambda expressions. Accordingly, we explored the null hypotheses $H2.1_0$ and $H2.2_0$ (Section 4). Tables 13 and 14 summarize the results of the regression analysis considering a significance level $<$ 0.05.

After performing the regression analysis, both models led to a p-value $>$ 0.05, w.r.t the SLOC metric. However, differently from the results of the first phase, the analyses led to a p-value $<$ 0.05 when considering the CC metric. Such results suggested that *cyclomatic complexity* can be used to estimate the impact on code comprehension after the introduction of lambda expressions. Therefore, the results confirmed our second hypothesis with respect to the *cyclomatic complexity* metric, being possible to estimate the effect on the readability metrics using the difference on the CC metric. We further detail these results in Section 6.

Figure 8. Pair of code snippet 499. Replacing a structural *for loop*.

```
public void rewind(int start) {
    currentTokenIndex = start;
    /** Remove any consume and lookahead attribute for any token with index
     * greater than start
     */
    for (Integer idx : inputTokenIndexes) {
        if (idx >= start) {
            indexToConsumeAttributeMap.remove(idx);
            lookaheadTokenIndexes.remove(idx);
        }
    }
}
```

(a)

```
public void rewind(int start) {
    currentTokenIndex = start;
    /** Remove any consume and lookahead attribute for any token with index
     * greater than start
     */
    inputTokenIndexes.stream().filter((idx) →(idx >= start)).map((idx) →{
        indexToConsumeAttributeMap.remove(idx);
        return idx;
    }).forEachOrdered((idx) →{
        lookaheadTokenIndexes.remove(idx);
    });
}
```

(b)

Table 13. Summary of 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)	0.0318	0.0220	1.44	0.1522
SLOCDiff	0.0038	0.0056	0.67	0.5034
CCDiff	-0.0623	0.0136	-4.59	0.0000

Table 14. Summary of 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)	0.0100	0.0161	0.62	0.5357
SLOCDiff	-0.0043	0.0041	-1.05	0.2975
CCDiff	-0.0253	0.0100	-2.54	0.0130

5.2 Qualitative Assessment

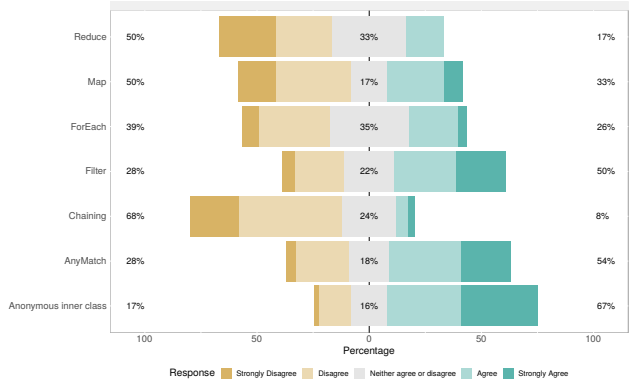
In the qualitative assessment, we report the results of a second survey with practitioners, to capture the perception of the developers about the impact on the readability of the code after applying transformations that introduce lambda expressions. These transformations had been recommended by automated tools only. We present the distribution of responses in the form of plots to build a broad perspective of the opinion of the respondents to every closed question. We then show the insights we got after conducting a thematic analysis of the open-ended questions, highlighting the participants' opinions with quotations and code examples.

5.2.1 The Impact of Introducing Lambda Expressions

In our second survey, our first question asked the opinion of the respondents about four sentences, which we use to understand the impact of introducing lambda expressions in the pairs of code snippets. We organize this section according to the sentences of the first question of the second survey.

The new code is easier to comprehend. The purpose of this sentence was to evaluate if the transformations to introduce lambda expressions (recommended by automated tools) improve program comprehension. Contrasting with the overall claims about the benefits of introducing lambda expressions (Gyori et al., 2013), we found that almost all types of transformations the automated tools suggest do not improve the readability of the programs.Interestingly, except for three types of transformations (*Anonymous Inner Class to Lambda*, *For loop to Any Match*, and *For loop to Filter*), the respondents most often did not agree that the introduction of lambda expressions makes the code easier to comprehend. Actually, according to Figure 9, 68% of the respondents stated that they did not agree that transformations involving the *chaining* of different stream operations improve program comprehension, and we observed the same trend for other typical recursive patterns (e.g., `map`, `reduce`, and `for each`).

Figure 9. Summary of the developers' answers to the sentence **The new code is easier to comprehend** of the survey.



It is worth to link these results to the answers to the open-ended question. That is, according to the participants, replacing an anonymous inner classes by a lambda expressions often improves program readability. Figure 10 shows an example of this particular type of transformation. After introducing the lambda expression, the code is more succinct because it removes some of the boilerplate code necessary to implement anonymous inner classes. Regarding the code snippet of Figure 10, one participant stated:

> "*(the code on the right is...) easier to read, usually lambda also makes the code cleaner and compact.*"

This comment suggests that this is a situation where the introduction of a lambda expressions improves program comprehension.

Differently, transformations involving *chaining* of the `stream` API methods received 68% of responses as either

Figure 10. Pair of code snippets 480. Replacing an *anonymous inner class* into *lambda expression.*

```
private ThrowingRunnable evaluateWithException(Exception e) {
    return new ThrowingRunnable() {
        public void run() throws Throwable {
            statement.nextException = e;
            statement.waitDuration = 0;
            failOnTimeout.evaluate();
        }
    };
}
```

(a)

```
private ThrowingRunnable evaluateWithException(Exception e) {
    return () →{
        statement.nextException = e;
        statement.waitDuration = 0;
        failOnTimeout.evaluate();
    };
}
```

(b)

Strongly disagree or *Disagree*, characterizing possible scenarios where the introduction of lambda expressions does not improve code comprehension. Another case involved transformations of *for loops* into `forEach` statements, which had 39% of negative answers (either *Strongly disagree* or *Disagree*. The type of transformations with the recursive patterns *map* and *reduce* received 50% of negative responses.

With respect to a transformation involving *chaining*, one of the respondents stated the following about the example of Figure 11.

> "*It's a bad example ... although I use lambdas a lot, I would never use them in exactly this way.*"

Considering the same example of code in Figure 11, another participant discussed that:

> "*(I would) almost never (execute this transformation). Transforming* for loops *into* `forEach` *statements with lambda expressions provides little benefit other than using a maybe slightly more concise syntax. "Readability" in my mind is such a subjective criterion that it is close to useless as a metric for making any decisions: someone coming from a functional language will find a map/filter/reduce pipeline easier to "read", and someone coming from a structured programming language will naturally tend towards nested loops.*"

This is an example of transformation that replaces `for each` statements by lambda expressions. According to the respondents, it does not improve program comprehension. Based on these results, we disclose that transformations of type Replacing anonymous inner class with lambda expressions, Replacing a for loop with the filter pattern and Replacing a for loop with the `AnyMatch` method improve code comprehension; while the transformations Replacing a for loop with a for-each statement, Replacing a for loop with the reduce pattern , Replacing a for loop with the map pattern, and Replacing a for loop with a *Chaining* of operators often do not improve program comprehension according to the developers' opinion.

Figure 11. Pair of code snippet 489. Replacing *Loop* to `forEach`, `filter` and `forEachOrdered`.

```
private void postConfigure() {
    List<Trigger> triggers = settings.getTriggers();
    for (Trigger trigger : triggers) {
        eventManager.addIvyListener(trigger, trigger.getEventFilter());
    }
    for (DependencyResolver resolver : settings.getResolvers()) {
        if (resolver instanceof BasicResolver) {
            ((BasicResolver) resolver).setEventManager(eventManager);
        }
    }
}
```

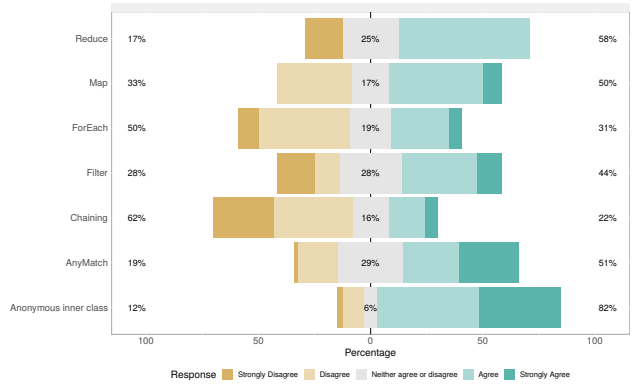
(a)

```
private void postConfigure() {
    List<Trigger> triggers = settings.getTriggers();
    triggers.forEach((trigger) →{
        eventManager.addIvyListener(trigger, trigger.getEventFilter());
    });
    settings.getResolvers()
      .stream()
      .filter((resolver) →(resolver instanceof BasicResolver))
      .forEachOrdered((resolver) →{
        ((BasicResolver) resolver).setEventManager(eventManager);
    });
}
```

(b)

The new code is more succinct and readable. The purpose of this sentence was to assess whether or not the introduction of lambda expressions makes the code more succinct and improves its readability. Figure 12 summarizes the results of the developers' responses to this particular sentence. In this case, we found a more positive tendency, and the transformations from anonymous inner class into lambda expressions and the transformations resulting in the `map`, `reduce`, `filter`, and `anyMatch` patterns present a leaning towards positive answers (*Agree* or *Strongly agree*). However, the assessment revealed that two types of transformations do not improve readability: transformations involving `forEach` and *chaining* of the `stream` API methods received more than 49% of negative responses (*Strongly Disagree* and *Disagree*).

Figure 12. Summary of the developers' answers to the sentence **The new code is more succinct and readable** of the survey.



The transformation in Figure 13 shows a scenario that replaces a `for each` statement by a call to the `forEach` method of the `stream` API. Although this is a straightforward situation where a developer might use a `forEach`, it does not improve the quality of the code, and most of the respondents considered that this particular scenario does not make the code more succinct and readable (more than 80% of the respondents are either neutral or does not agree that his transformation brings these benefits). Regarding this pair of code snippets, one of the respondents clearly stated this perception.

> "*(this) transformation does not improve readability and makes debugging more difficult.*"

Figure 13. Pair of code snippet 504. Replacing a *for loop* by a `forEach` pattern.

```
public ContextConfigurator updatedWith(Properties newProperties) {
  for (String key : newProperties.stringPropertyNames()) {
    withParameter(key, newProperties.getProperty(key));
  }
  return this;
}
```

(a)

```
public ContextConfigurator updatedWith(Properties newProperties) {
  newProperties.stringPropertyNames().forEach((key) →{
    withParameter(key, newProperties.getProperty(key));
  });
  return this;
}
```

(b)

Differently, Figure 14 shows an example of transformation that makes the code more succinct and readable, according to the opinion of the respondents. In this case, more than 80% of the answers were either neutral or present a leaning towards the agreement that the resulting code is more succinct and readable.

Altogether, from these observations, we argue that transformations replacing *for loops* by a `forEach` method call and the composition of stream operations (*sec:chaining*) do not improve readability or make the code more succinct. On the other hand, the other types of transformations have shown benefits regarding code readability.

Figure 14. Pair of code snippet 465. Replacing `Anonymous inner class`.

```
public FitNesse(FitNesseContext context) {
  this.context = context;
  RejectedExecutionHandler handler = new RejectedExecutionHandler() {
    @Override
    public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {
      LOG.log(WARNING, "Could not handle request. Thread pool ...");
    }
  };
  //...
}
```

(a)

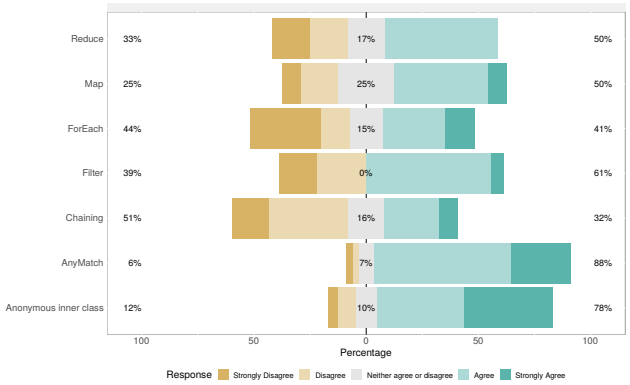
```
public FitNesse(FitNesseContext context) {
  this.context = context;
  RejectedExecutionHandler handler = (Runnable r, ThreadPoolExecutor e) →{
      LOG.log(WARNING, "Could not handle request. Thread pool ...");
  };
  //...
}
```

(b)

The intention of using a lambda expression in the new code is clear. The purpose of this question was to investigate whether or not developers are able to understand the motivation for using the lambda expressions introduced in

the new code. Figure 15 summarizes the results of the developers responses to this question. Similarly to the previous sentence, we found a more negative leaning when we considered the transformations that replace a `for loop` by a call to the `forEach` method and transformations that introduce a *chaining* of `stream` operations. The remaining types of transformations seemed to make clear the intention of using either a lambda expression instead of an anonymous inner class or a recursive pattern (e.g., `filter`, `anyMatch`, `map` or `reduce`) instead of a *for loop*.

Figure 15. Summary of the developers' answers to the sentence **The intention of using a lambda expression in the new code is clear** of the survey.



Transformations introducing a call to the `forEach` method received 44% of negative (*Strongly Disagree* or *disagree*) responses. This suggests a neutral opinion regarding the *clear intention* of introducing a lambda expression. Figure 16 shows an example of code that replaces a *for loop* by the `forEach` pattern, where 66% of the respondents considered unclear the intention of the code. In particular, a participant stated that:

> "*(I would never) perform this transformation. The* for loop *makes it clear and explicit that we are iterating over the elements in the collection—it is a fundamental part of the language that we all understand. The (use of) lambda expression does not.*"

Figure 17 shows an example of transformation that makes the intention of the code clearer. This transformation replaces a *for loop* by a call to the `anyMatch` method, and 88% of the respondents assigned either a neutral or a positive answer (*Agree* or *Strongly agree*) with respect to the *clear intention* of using a lambda expression in this example. A respondent also claimed that:

> "*...The new code is more elegant and makes the intention of finding some occurrence where the condition is true clearer.*"

Altogether, from these observations, we argue that transformations replacing *for loops* by calls to the `forEach` method and the composition of stream operators (*chaining*) do not make clear the intention of introducing lambda expressions. On the other hand, the other types of transformations

Figure 16. Pair of code snippets 502. Replacing a *for loop* with the `forEach` pattern.

```
protected Map<Object, Object> cSort(List<?> list, int col) {
  TypeAdapter a = columnBindings[col].adapter;
  Map<Object, Object> result = new HashMap<>(list.size());
  for (Object row : list) {
    try {
      a.target = row;
      Object key = a.get();
      bin(result, key, row);
    }
    catch (Exception e) {
      // surplus anything with bad keys, including null
      surplus.add(row);
    }
  }
  return result;
}
```

(a)

```
protected Map<Object, Object> cSort(List<?> list, int col) {
  TypeAdapter a = columnBindings[col].adapter;
  Map<Object, Object> result = new HashMap<>(list.size());
  list.forEach((row) →{
      try {
          a.target = row;
          Object key = a.get();
          bin(result, key, row);
      }
      catch (Exception e) {
          // surplus anything with bad keys, including null
          surplus.add(row);
      }
  });
  return result;
}
```

(b)

Figure 17. Pair of code snippet 548. Replacing a *for loop* with the `anyMatch` pattern

```
private static boolean isAssignableToAnyOf(Class<?>[] typeArray, Object target) {
    for (Class<?> type : typeArray) {
        if (type.isAssignableFrom(target.getClass())) {
            return true;
        }
    }
    return false;
}
```

(a)

```
private static boolean isAssignableToAnyOf(Class<?>[] typeArray, Object target) {
    return typeArray.stream()
        .anyMatch(type →type.isAssignableFrom(target.getClass()));
}
```

(b)

have shown benefits, making it clear the intention of replacing *anonymous inner classes* with lambda expressions and the use of other recursive patterns (`filter`, `anyMatch`, `map`, and `reduce`).

The new code is harder to debug. The goal of this sentence was to assess whether or not the introduction of lambda expressions makes the code more difficult to debug. The results in Figure 18 show that practically all types of transformations present the side effect of hindering the task of debugging, apart from the transformations that replace anonymous inner classes by lambda expressions.

Transformations involving calls to the `filter` and `chaining` methods of the `stream` API received more than 70% of negative responses—that is, respondents either *Agree* or *Strongly agree* that the transformations make the code harder to debug. Differently, transformations that replace

anonymous inner classes by lambda expressions received 53% of positive answers (respondents consider that this kind of transformation does not hinder debugging activities).

Figure 18. Summary of the developers' answers to the sentence **The new code is harder to debug** of the survey.

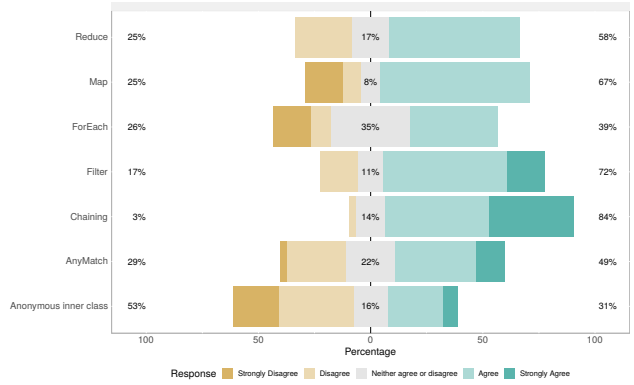


Figure 19 shows an example of a transformation that introduces a `forEach` statement. In this case, 88.33% of the respondents were either neutral or presented a positive feeling that this transformation does not hinder debugging tasks. Interesting, one participant claimed that this transformation made the code harder to debug (due to obfuscating the types of variables), although he/she was still leaning towards considering the transformation beneficial.

> "*Obfuscating the types of the variables used makes the code easier to change, but at the same time may make it harder to debug. I would still perform the transformation though.*"

Figure 19. Pair of code snippet 510. Replacing `loop` to `forEach` pattern.

```
public synchronized void addError(Test test, Throwable e) {
    fErrors.add(new TestFailure(test, e));
    for (TestListener each : cloneListeners()) {
        each.addError(test, e);
    }
}
```

(a)

```
public synchronized void addError(Test test, Throwable e) {
    fErrors.add(new TestFailure(test, e));
    cloneListeners().forEach((each) -> {
        each.addError(test, e);
    });
}
```

(b)

Figure 20 shows an example of transformation that also makes the code hard to debug (more than 85% of the respondents either *Agree* or *Strongly agree* that this transformation hinders debugging tasks). However, in the opinion of a developer, an improvement in the transformation could actually make the resulting code easier to debug.

> "*Yes (I would perform this transformation), in a hurry, but with a minute more time I'd extract the filter into its own function. However, the suggested refactoring is in itself valuable because it does bring out the important part. If an automated tool did this to a whole codebase, it would make debugging easier, especially for junior developers.*"

Figure 20. Pair of code snippet 491. Replacing anonymous inner class with lambda expressions.

```
public File[] getConfigurationResolveReportsInCache(final String resolveId) {
    final String prefix = resolveId + "–";
    final String suffix = ".xml";
    return getResolutionCacheRoot().listFiles(new FilenameFilter() {
        public boolean accept(File dir, String name) {
            return name.startsWith(prefix) && name.endsWith(suffix);
        }
    });
}
```

(a)

```
public File[] getConfigurationResolveReportsInCache(final String resolveId) {
    final String prefix = resolveId + "–";
    final String suffix = ".xml";
    return getResolutionCacheRoot().listFiles((dir, name) →
        name.startsWith(prefix) && name.endsWith(suffix));
}
```

(b)

In summary, from these observations, we argue that evolving a legacy code to use the `stream` API and lambda expressions often makes the resulting code harder to debug. This undesired side effect does not happen in the case of transformations from anonymous inner classes into lambda expressions.

5.2.2 How often would you perform this type of transformation?

The purpose of this question was to assess how often developers would perform the set of 98 transformations we explore during the survey. Interesting, besides the possible side effect of hindering debugging activities, respondents presented a positive tendency to accept 72% of the transformations in our dataset—respondents rejected 22% of the transformations and were neutral with respect to 6% of the transformations. Nonetheless, when we discarded the transformations involving anonymous inner classes, the number of transformations that the respondents would accept dropped from 72% to 44.44%, and the respondents would reject 50% of the transformations.

Figure 21 summarizes the responses to this question, which presents options related to frequency (from *Never* to *Always*). It is possible to observe that the respondents would not perform some of the transformations. For instance, the respondents would never or rarely replace a *for loop* by a call to the `forEach` method in 50% of the scenarios. We found a similar result when considering transformations that introduce the `map` recursive pattern. Differently, the respondents stated they will either *Often* or *Always* perform transformations replacing *for loops* by a call to the `anyMatch` method (61%) and inner classes by lambda expressions (60%). Table 15 presents a different perspective about the answers to this question, without splitting them using the type of the transformations.

Figure 21. Summary of the developers' answers to the question **How often would you perform this type of transformation?** of the survey.

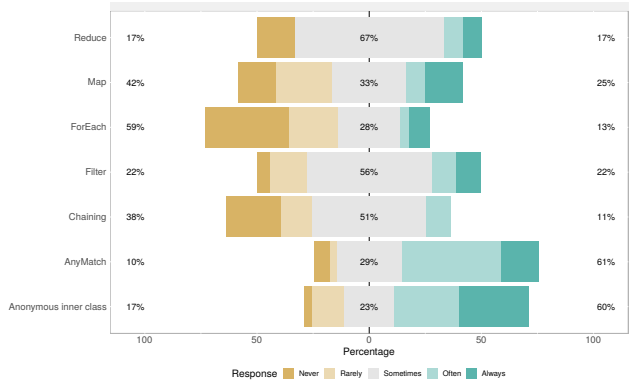


Table 15. Summary of the answers for the question *How often would you perform this type of transformation?*

S2Q2	Answers	Percentage	Cum. Percentage
Never	55	8.65%	8.65%
Rarely	92	14.5%	23.5%
Sometimes	173	27.2%	50.35%
Often	160	25.2%	75.5%
Always	156	24.5%	100.0%
Total	636	100.0%	

5.2.3 How important is the automated support for this kind of transformation?

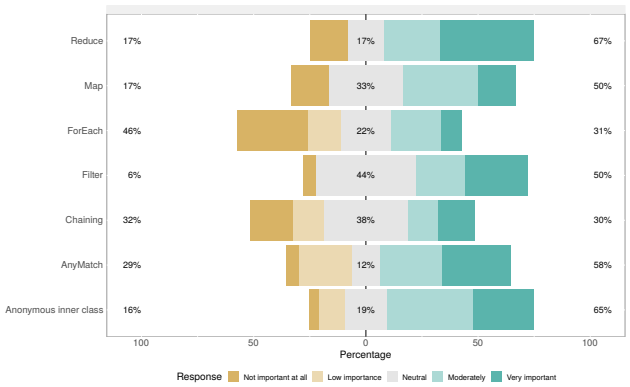
The purpose of this question was to assess the importance of using tools to perform transformations that introduce *lambda expressions*. Figure 22 summarizes the results for this question, where the options range from *Not important at all* to *Very Important*. We can observe in the figure that respondents considered the support of automated tools either *Moderately Important* or *Very Important* to apply the transformation, in more than 50% of the cases. This might indicate that developers prefer to perform these transformations using some code refactoring tool. However, transformations introducing the `forEach` recursive pattern received most of the responses between *Not important at all* and *Low important*, which perhaps supports that this particular kind of transformation does not improve the source code. Finally, the transformation classified as Replacing a for loop with a *Chaining* of operators received most responses in Neutral (38%).

Based in these results, we can argue that developers consider worth the use of refactoring tools to introduce lambda expressions and rejuvenate Java programs. However, there is some room for improving these tools, as we discuss possible scenarios in the next section.

5.2.4 Synthesis of the Responses to the Open-ended Question

In this section we present a synthesis of answers to the open-ended question of our second survey, using the *thematic analysis* procedures we detailed in Section 3. We found three **recurrent** themes that might explain the reasons for accepting a transformation: *More Succinct Code*, *Easier to Understand*, and *Clear Code Intention*. We also identified three **recurrent** themes that might justify why a given transformation should not be applied: *Small Benefit*, *Harder to Understand*, and *Wrong Scenario* for using a lambda expression. Finally, several answers claim that the transformations could be improved (the *Need Improvements* theme that appears in transformations marked either as accepted or rejected). Several answers provided an alternative to the modified version of the code (often using a textually description, but in a few cases, the participants also shared as code example using a Gist[4]).

Figure 22. Summary of the developers' answers to the question **How important is the automated support for this kind of transformation?** of the survey.



Most recommendations to improve the resulting code (i.e., the code after applying a transformation) relate to the source code format, e.g.: "*No need of curly braces and semicolon on the second statement*" and "*I would always perform this transformation, but I would use line breaks and filters to make the code more readable*". Perhaps, refactoring engines that introduce lambda expressions could benefit from advanced code format tools (e.g., the approach by Parr and Vinju (2016)). Other possible improvements are trickier, which might indicate the need to follow a careful code review process after applying code transformations (Carvalho et al., 2020). For instance, one of the participants argued that:

> "*[...] streams should produce collections as results, not populate them as side-effects. If we fixed that, and broke to a new line before each transformation or filter, then I think it would be OK.*"

Other possible improvements stress the use of the type inference mechanism: "*I don't think you need to specify (File file), do you? You could just say "file" and let the type get inferred [, right]? Unless CollectionUtils.select is overloaded and takes multiple different functional types.*" We found that the transformation engines of NetBeans IDE and RJTL do not explore the type inference mechanism in their refactoring recommendations. Participants also suggested that the introduction of lambda expressions brings small benefits, and, as such, they would rarely change a legacy code that is working just to introduce new language constructs or idioms.

> "*I would not rewrite legacy code to introduce a lambda expression in this way, unless the inner code itself would have to be rewritten.*"

Table 16. Features that point to code improvements after introducing lambda expressions.

Themes	Frequency	Description	Representative Examples	Participants
More Succinct Code	19	They are transformations to introduce lambda expressions that make the code more succinct.	”yes, perfect case for lambda, short, clear”; ”Yes, I would because nowadays languages have improved their syntax to provide a better and easy code to developers make their softwares, Java 8 introduced Lambda, where you can write less code and do more.”; ”I would sometimes make this change, but not always because it is only making the code more succinct”;	P203, P285, P749
Easier to understand	15	They are transformations to introduce lambda expressions that make the code more comprehensibly.	”Yes. The new code, besides looking cleaner, is also really easier to read and comprehend.”; ”Yes, code readability was a factor”; ”Easier to read, usually lambda also makes the code cleaner and compact”;	P803, P334, P337
Clear Code Intention	14	They are transformations to introduce lambda expressions that make the code more clear.	”Yes, since it looks more ”straight forward”, and it makes the code itself cleaner”; ”I would do it because it’s easier to write and the code gets cleaner.”; ”Yes, absolutely, clearer intent, more expressive, easier to read and comprehend.”;	P803, P635, P229

Table 17. Features that point to code worsening after introducing lambda expressions.

Themes	Frequency	Description	Representative Examples	Participants
Small benefit	27	They are transformations to introduce lambda expressions that have little or no benefit.	”No. The benefit isn’t big enough to perform the transformation.”; ”I believe, in this example, the transformation is a small part of the method and it does not influence positively or negatively at all the legibility of the method.”; ”I consider both versions to be similar”;	P268, P138, P166
Harder to understand	5	They are transformations to introduce lambda expressions that make code less comprehensibly.	”This is still pretty hard to read and understand on account of a) the hard cast of the lambda to Callable<Object>, which seems weird - is this necessary? Isn’t it at least a Callable<T>? b) Why a ”checkThat” method is calling ”checkSucceeds” which seems a little like jumping to a conclusion.”; ”Maybe not a complex return on one line”; ”I would never perform this transformation. The for loop makes it clear an explicit that we are iterating over the elements in the collection - it is a fundamental part of the language that we all understand.”;	P229, P203, P583
Wrong scenario	5	They are transformations to introduce lambda expressions that shouldn’t be done.	”Since this is a void method it will, by definition, never be truly functional. Splitting the original code into a map – with a side effect, no less! – and a terminal operation with forEach construct does not really improve anything in my mind.”; ”I tend to avoid try-catch in lambda expressions. I don’t think it’s bad to do so, but I personally don’t do it, even if it means using an anonymous inner class.”;	P694, P547

Table 16 and Table 17 summarize the frequency of the recurrent themes. As a future work, our goal is to consider the answers to this open-ended question to improve the RJTL implementation. All code snippets and datasets we used in our research are available in the paper’s companion website [5].

6 Discussion

As explained in the previous section, we found conflicting results in our research. In the first phase, the models for estimating readability diverge from one another. That is, the Buse and Weimer (2010) model suggests that when a developer introduces a lambda expression into Java legacy method, the readability of the method **decreases**. Differently, the model of Posnett et al. (2011) suggests that the introduction of lambda expressions does not impact program comprehension in the first phase. Contrasting, in the second phase, both models suggest that the introduction of lambda expressions decreases program comprehension. The main difference between the two phases is that the second one only consider transformations suggested by automated tools. Perhaps, manual transformations fix some problems related to readability.

Nonetheless, the results of the qualitative assessments with practitioners suggest that the introduction of lambda expressions improves program comprehension in particular cases. For instance, the replacement of anonymous inner classes by lambda expressions often improve readability—according to the results of our surveys. Other scenarios that the introduction of lambda expressions might be positive are the replacement of *for loops* with simple recursive patterns like `filter` and `anyMatch`. We believe that these conflicting results are partially due to the limitations of both models on identifying improvements in finer-grained transformations. Considering the results of both quantitative and qualitative studies, we answer our research questions in Section 6.1 and present some lessons learned in Section 6.2. Finally, we present some threats to the validity of our study in Section 6.3.

6.1 Answers to The Research Questions

When using a mixed-methods approach, the best scenario occurs in situations where the results of a quantitative studies support the findings and explains the results of the qualitative ones (or vice-versa). Considering Table 18, which combines the results of the quantitative and qualitative assessment for the transformations that replace anonymous inner classes with lambda expressions, it is possible to observe differences between the outcomes of both readability models and the developers perceptions of code comprehension.

[4] Gist is a GitHub feature that allow developers to share code

[5] https://waltim.github.io/jserd.html

We are in favor of the results of the qualitative study. Therefore, considering our first research question (*Does the use of lambda expressions improve program comprehension?*), our findings revealed that refactoring a legacy code to introduce lambda expression improves program comprehension in the specific scenarios we discussed earlier.

Table 18. Number of code snippets that increased readability, decreased readability and unchanged readability; after replacing anonymous inner classes with lambda expressions by the tools.

Evaluator	Increased	Decreased	Unchanged
Buse and Weimer	23	32	3
Posnett et al.	11	43	4
Developers	**51**	3	4

After these results, we investigated whether the code complexity metrics (SLOC and CC), independently, could predict if a transformation of a legacy code to introduce lambda expressions improves the readability of the code. To perform this investigation, we calculated the differences in SLOC (Δs) and CC (Δcc) metrics, considering the code snippets before and after the introduction of lambda expressions. After that, we ran the Pearson's correlation test (Mukaka, 2012), to assess whether these differences correlate with possible improvements in program comprehension according to the survey respondents. We found that (Δcc) has no relation to the answers of developers about comprehension. On the other side, the (Δs) presents a moderate correlation ($\rho = 0.5324$ and p-value < 0.05). Such results revealed that the greater the reduction of lines after the introduction of lambda expressions, the better the comprehension of the code according to the developers opinion—independently of reducing the cyclomatic complexity or not. Therefore, tool developers could use SLOC to automatic learn good situations to suggest transformations that introduce lambda expressions.

Regarding the second research question (*Does the introduction of lambda expressions reduce source code complexity?*), after assessing the impact of introducing lambda expressions in 158 pairs of code snippets (66 of the first phase and 92 from the second phase of this research), we found that introducing lambda expressions (a) reduces the size of the code (SLOC) in 70% of the cases and (b) reduces the cyclomatic complexity in 40% of the cases. Only in a few cases, the introduction of lambda expressions increased SLOC. We did not find any case in which a transformation increases cyclomatic complexity. Considering our third research question (*What are the most suitable situations to refactor code to introduce lambda expressions?*), we found that replacing anonymous inner class by a lambda expressions might be considered the *killer application* to introduce lambda expressions in legacy Java code. In addition, scenarios replacing *for loops* having internal conditional with an `anyMatch` operator often improved the readability of the code and makes the intention of using the lambda expression more clear. Differently, just replacing a simple `for` over a collection statement with a `collections.forEach()` did not bring any benefit, according to the participants of our surveys. We also found that the chaining of `stream` methods and the introduction of recursive patterns (e.g., `filter` and `map`) hinders debugging activities according to the developers.

Regarding our fourth research question (*How do practitioners evaluate the effect of introducing lambda expressions into a legacy code?*), developers agreed that the introduction of lambda expressions improve the quality of the code (in particular when removing the boilerplate code related to anonymous inner classes), though it might introduce some challenges to debugging activities in general. Developers would actually accept most of the RJTL, NetBeans, and IntelliJ transformations (72%), and they considered worth the existence of automated support to introduce lambda expressions and thus rejuvenate Java legacy code.

Finally, with respect to our last research question (*What is the practitioners' opinion about the recommendations from automated tools to introduce lambda expressions?*), the results suggested that the use of automated tools to rejuvenate Java programs is promising. Again, considering only recommendations from NetBeans IDE, RJTL, and IntelliJ IDE, developers agreed that transformations replacing anonymous inner class by lambda expressions improve program comprehension. Still, the feedback from the participants revealed several weaknesses of these tools, and thus we found some space to improve these refactoring engines, as we discuss in the next section.

6.2 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 also fail when evaluating general transformations such as more popular refactorings. Nonetheless, both models are sensitive for code formatting decisions, including the number of blank characters. Similar conclusions have been reported in a recent research work Fakhoury et al. (2019).

Recommendations for Refactoring Tools. We found that transforming anonymous inner class into lambda expressions is the scenario that brings more benefits for code comprehension. We also found that replacing *for loops* having an internal conditional by an `anyMatch` and `filter` patterns improves the code readability. Nonetheless, we consider that it is not recommended to blindly apply automatic transformations from simple `for loop` statements into a `collections.forEach()` statement. This kind of transformations does not improve code readability. Several features might also help to identify the situations where introducing a lambda expression do not improve the code. For example, according to the participants, we should avoid combining the functional and imperative styles in the same method. Similarly, several transformations led to pieces of code with a wrong indentation (e.g., comprising long lines or unnecessary curly braces). According to the practitioners, some recommendations decreased the readability of the code due to indentation issues.

6.3 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 used a sample from real scenarios that introduce lambda expressions in legacy code, this sample might not correspond to a representative population that would be recommended to conclude our quantitative assessment. We evaluated nine pairs of code snippets in the first survey. To circumvent such a threat, we replicated the study and evaluated 92 pairs of code snippets. This number is similar to the number of code snippets evaluated in a previous study (Posnett et al., 2011).

The second threat is related to external validity. Initially, our research participants belonged to a relatively small group of professional developers, who despite having great experience in Java, were a small group of developers in our cycle. During the replication of the study, we were able to significantly increase the number of participants from different locations in the world. We believe that, with this variety of participants, our results became more reliable, allowing us to generalize our findings to this population.

Finally, we could have used other models to estimate readability, which have been previously discussed in the literature (Scalabrino et al., 2016). However, we only found an implementation of one of these models, the one by Buse and Weimer (2010). We also implemented the computation for an additional model by Posnett et al. (2011), but it would be difficult to provide implementations for all models available in the literature.

7 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 (Buse and Weimer, 2010; Posnett et al., 2011), and found conflicting results. Both models (Posnett et al., 2011) and (Buse and Weimer, 2010) suggested that the introduction of lambda expressions does not improve the comprehensibility of the source code. Differently, the results of the qualitative studies (surveys with practitioners) indicated that the introduction of lambda expressions in legacy code improves code comprehension in particular cases (particularly when replacing anonymous inner classes by lambda expressions). After considering these conflicting results, we argue that (a) this kind of source code transformation improves software readability for specific scenarios and (b) we need more advanced models to understand the benefits on program comprehension after applying finer-grained program transformations.

Acknowledgements

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

Alqaimi, A., Thongtanunam, P., and Treude, C. (2019). Automatically generating documentation for lambda expressions in java. In *Proceedings of the 16th International Conference on Mining Software Repositories*, MSR '19, pages 310–320, Piscataway, NJ, USA. IEEE Press.

Avidan, E. and Feitelson, D. G. (2017). Effects of variable names on comprehension an empirical study. In Scanniello, G., Lo, D., and Serebrenik, A., editors, *Proceedings of the 25th International Conference on Program Comprehension, ICPC 2017, Buenos Aires, Argentina, May 22-23, 2017*, pages 55–65. IEEE Computer Society.

Baggen, R., Correia, J. P., Schill, K., and Visser, J. (2012). Standardized code quality benchmarking for improving software maintainability. *Software Quality Journal*, 20(2):287–307.

Buse, R. P. L. and Weimer, W. (2010). Automatically documenting program changes. In Pecheur, C., Andrews, J., and Nitto, E. D., editors, *ASE 2010, 25th IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium, September 20-24, 2010*, pages 33–42. ACM.

Carvalho, A., Luz, W. P., Marcilio, D., Bonifácio, R., Pinto, G., and Canedo, E. D. (2020). C-3PR: A bot for fixing static analysis violations via pull requests. In Kontogiannis, K., Khomh, F., Chatzigeorgiou, A., Fokaefs, M., and Zhou, M., editors, *27th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2020, London, ON, Canada, February 18-21, 2020*, pages 161–171. IEEE.

Dantas, R., Carvalho, A., Marcilio, D., Fantin, L., Silva, U., Lucas, W., and Bonifácio, R. (2018). Reconciling the past and the present: An empirical study on the application of source code transformations to automatically rejuvenate java programs. In Oliveto, R., Penta, M. D., and Shepherd, D. C., editors, *25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018*, pages 497–501. IEEE Computer Society.

dos Santos, R. M. and Gerosa, M. A. (2018). Impacts of coding practices on readability. In Khomh, F., Roy, C. K., and Siegmund, J., editors, *Proceedings of the 26th Conference on Program Comprehension, ICPC 2018, Gothenburg, Sweden, May 27-28, 2018*, pages 277–285. ACM.

Fakhoury, S., Roy, D., Hassan, S. A., and Arnaoudova, V. (2019). Improving source code readability: Theory and practice. In *Proceedings of the 27th International Conference on Program Comprehension*, ICPC '19, pages 2–12, Piscataway, NJ, USA. IEEE Press.

Favre, J.-M., Lämmel, R., Schmorleiz, T., and Varanovich, A. (2012). 101companies: A community project on software technologies and software languages. In Furia, C. A. and Nanz, S., editors, *Objects, Models, Components, Patterns*, pages 58–74, Berlin, Heidelberg. Springer Berlin Heidelberg.

Godfrey, M. W. and German, D. M. (2008). The past, present, and future of software evolution. In *2008 Frontiers of Software Maintenance*, pages 129–138.

Gopstein, D., Iannacone, J., Yan, Y., DeLong, L., Zhuang, Y., Yeh, M. K.-C., and Cappos, J. (2017). Understanding misunderstandings in source code. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 129–139, New York, NY, USA. ACM.

Gyori, A., Franklin, L., Dig, D., and Lahoda, J. (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, pages 543–553, New York, NY, USA. ACM.

Khatchadourian, R., Tang, Y., Bagherzadeh, M., and Ahmed, S. (2019). Safe automated refactoring for intelligent parallelization of java 8 streams. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, pages 619–630, Piscataway, NJ, USA. IEEE Press.

Landman, D., Serebrenik, A., Bouwers, E., and Vinju, J. J. (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):589–618.

Lehman, M. M. and Ramil, J. F. (2001). Rules and tools for software evolution planning and management. *Annals of software engineering*, 11(1):15–44.

Lott, S. F. (2018). *Functional Python Programming: Discover the power of functional programming, generator functions, lazy evaluation, the built-in itertools library, and monads*. Packt Publishing Ltd.

Lucas, W., Bonifácio, R., Canedo, E. D., Marcilio, D., and Lima, F. (2019). Does the introduction of lambda expressions improve the comprehension of java programs? In do Carmo Machado, I., Souza, R., Maciel, R. S. P., and Sant'Anna, C., editors, *Proceedings of the XXXIII Brazilian Symposium on Software Engineering, SBES 2019, Salvador, Brazil, September 23-27, 2019*, pages 187–196. ACM.

Mazinanian, D., Ketkar, A., Tsantalis, N., and Dig, D. (2017). Understanding the use of lambda expressions in java. *Proc. ACM Program. Lang.*, 1(OOPSLA):85:1–85:31.

Mukaka, M. M. (2012). A guide to appropriate use of correlation coefficient in medical research. *Malawi medical journal*, 24(3):69–71.

Overbey, J. L. and Johnson, R. E. (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, pages 493–502, New York, NY, USA. ACM.

Parr, T. and Vinju, J. J. (2016). Towards a universal code formatter through machine learning. In van der Storm, T., Balland, E., and Varró, D., editors, *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering, Amsterdam, The Netherlands, October 31 - November 1, 2016*, pages 137–151. ACM.

Pennington, N. (1987). Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology*, 19(3):295 – 341.

Posnett, D., Hindle, A., and Devanbu, P. T. (2011). A simpler model of software readability. In van Deursen, A., Xie, T., and Zimmermann, T., editors, *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*, pages 73–82. ACM.

Riaz, M., Mendes, E., and Tempero, E. (2009). A systematic review of software maintainability prediction and metrics. In *2009 3rd International Symposium on Empirical Software Engineering and Measurement*, pages 367–377.

Scalabrino, S., Linares-Vásquez, M., Poshyvanyk, D., and Oliveto, R. (2016). Improving code readability models with textual features. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, pages 1–10.

Shrestha, N., Botta, C., Barik, T., and Parnin, C. (2020). Here we go again: Why is it difficult for developers to learn another programming language? In *Proceedings of the 42nd International Conference on Software Engineering, ICSE*.

Silva, D., Tsantalis, N., and Valente, M. T. (2016). Why we refactor? confessions of github contributors. In Zimmermann, T., Cleland-Huang, J., and Su, Z., editors, *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, pages 858–870. ACM.

Storey, M. D., Wong, K., and Müller, H. A. (2000). How do program understanding tools affect how programmers understand programs? *Sci. Comput. Program.*, 36(2-3):183–207.

Stroustrup, B. (2013). *The C++ Programming Language*. Addison-Wesley Professional, 4th edition.

Tilley, S. R., Paul, S., and Smith, D. B. (1996). Towards a framework for program understanding. In *WPC '96. 4th Workshop on Program Comprehension*, pages 19–28.

Tsantalis, N., Mazinanian, D., and Rostami, S. (2017). Clone refactoring with lambda expressions. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 60–70.

Urma, R.-G., Fusco, M., and Mycroft, A. (2014). *Java 8 in Action: Lambdas, Streams, and functional-style programming*. Manning Publications Co.

von Mayrhauser, A. and Vans, A. M. (1995). Program comprehension during software maintenance and evolution. *IEEE Computer*, 28(8):44–55.

Wilcoxon, F. (1945). Individual comparisons by ranking methods. *Biometrics Bulletin (JSTOR)*, 1(6):80–83.

A Taxonomy of Lambda Expression Transformations

This appendix introduces a simple taxonomy used to classify the lambda expression transformations. For each member of the taxonomy, we present a brief description and an example.

Replacing anonymous inner class with lambda expressions

A developer might use this transformation to convert an anonymous inner class into a lambda expression. Figure 23 shows an example of this transformation.

Figure 23. Pair of code snippet 551. Replacing the *Anonymous Inner Class*.

```
public void runTest() {
    runBeforesThenTestThenAfters(new Runnable() {
        public void run() {
            runTestMethod();
        }
    });
}
```

(a)

```
public void runTest() {
    runBeforesThenTestThenAfters(()→ { runTestMethod();});
}
```

(b)

Replacing a for loop with the map pattern

A developer might use this transformation to convert a *for loop* into a `map` recursive pattern of the `stream` API. Figure 24 shows an example of this transformation.

Figure 24. Pair of code snippet 495. Replacing `loop` to `Map` pattern.

```
public void draw(Graphics2D g) {
    for (Color c : shapes.keySet()) {
        g.setColor(c);
        g.draw(shapes.get(c));
    }
}
```

(a)

```
public void draw(Graphics2D g) {
    shapes.keySet().stream().map((c) →{
        g.setColor(c);
        return c;
    }).forEachOrdered((c) →{
        g.draw(shapes.get(c));
    });
}
```

(b)

Replacing a for loop with the reduce pattern

A developer might use this transformation to convert a *for loop* into a `reduce` pattern of the `stream` API. Figure 25 shows an example of this transformation. In this example, there is a composition between a `map` and a `reduce`, though the goal is to reduce a collection of test classes into the number of test methods.

Figure 25. Pair of code snippet 513. Replacing `Loop` to `reduce`.

```
public int countTestCases() {
    int count = 0;
    for (Test each : fTests) {
        count += each.countTestCases();
    }
    return count;
}
```

(a)

```
public int countTestCases() {
    int count = 0;
    count = fTests.stream()
                  .map((each) →each.countTestCases())
                  .reduce(count, Integer::sum);
    return count;
}
```

(b)

Replacing a for loop with a for-each statement

A developer might use this transformation to convert a *for loop* into a `forEach` statement. Figure 26 shows an example of this transformation. Respondents of our survey do not consider that this kind of transformation improves the quality of the code.

Figure 26. Pair of code snippet 500. Replacing `Loop` to `forEach` pattern.

```
public List<String> getPotentialFixtureClassNames(Set<String> elements) {
  List<String> candidateClassNames = new ArrayList<>();
  if (!isFullyQualified()) {
    for (String packageName : elements) {
      addBlahAndBlahFixture(packageName + ".", candidateClassNames);
    }
  }
  addBlahAndBlahFixture("", candidateClassNames);
  return candidateClassNames;
}
```

(a)

```
public List<String> getPotentialFixtureClassNames(Set<String> elements) {
  List<String> candidateClassNames = new ArrayList<>();
  if (!isFullyQualified()) {
      elements.forEach((packageName) →{
          addBlahAndBlahFixture(packageName + ".", candidateClassNames);
      });
  }
  addBlahAndBlahFixture("", candidateClassNames);
  return candidateClassNames;
}
```

(b)

Replacing a for loop with the filter pattern.

A developer might use this transformation to convert a *for loop* into the `filter` recursive pattern of the `stream` API. Figure 27 shows an example of this transformation. Respondents in our survey consider that this type of transformation improves the quality of the code.

Figure 27. Pair of code snippet 547. Replacing *loop* to Replacing a for loop with the filter pattern recursive pattern.

```
public ClassPath(List<ClassPath> paths) {
    this.elements = new ArrayList<>();
    this.separator = paths.get(0).getSeparator();
    for (ClassPath path : paths) {
        for (String element : path.getElements()) {
            if (!elements.contains(element)) {
                elements.add(element);
            }
        }
    }
}
```

(a)

```
public ClassPath(List<ClassPath> paths) {
  elements = path.getElements().stream()
    .filter(e →!elements.contains(e)).collect(Collectors.toList());}
}
```

(b)

Replacing a for loop with the `AnyMatch` method.

A developer might use this transformation to convert a *for loop* and conditional `if` into the `anyMatch` method. Figure 28 shows an example of this transformation. Respondents in our survey consider that this type of transformation improves the quality of the code.

Figure 28. Pair of code snippet 555. Replacing a for loop with the `AnyMatch` pattern.

```
private boolean isOverridenWithoutAnnotation(Method[] methods,
        Method superclazzMethod, Class<? extends Annotation> annotation) {
    for (Method method : methods) {
        if (isMethodOverride(method, superclazzMethod)
                && (method.getAnnotation(annotation) == null)) {
            return true;
        }
    }
    return false;
  }
```

(a)

```
private boolean isOverridenWithoutAnnotation(Method[] methods, Method
    ↪ superclazzMethod, Class<? extends Annotation> annotation) {
  return methods.stream().anyMatch(method →isMethodOverride(method,
      ↪ superclazzMethod) && (method.getAnnotation(annotation) == null))
      ↪ ;
}
```

(b)

Replacing a for loop with a *Chaining* of operators.

A developer might use this transformation to convert a *for loop* into the `Chaining` operators. Figure 29 shows an example of this transformation where is addition a sequence of distinct patterns (`Map` and `Filter`) followed by `forEachOrdered` statement.

Figure 29. Pair of code snippet 493. Replacing `Loop` to *chain* of *stream* operators.

```
private void rememberAllOpenedDocuments() {
    final List<String> docPath = new ArrayList<String>();
    for (XJWindow window : XJApplication.shared().getWindows()) {
        final XJDocument document = window.getDocument();
        if(XJApplication.handlesDocument(document)) {
            docPath.add(document.getDocumentPath());
        }
    }
    AWPrefs.setAllOpenedDocuments(docPath);
}
```

(a)

```
private void rememberAllOpenedDocuments() {
    final List<String> docPath = new ArrayList<String>();
    XJApplication.shared().getWindows().stream().map((window) →window.
        ↪ getDocument()).filter((document) →(XJApplication.
        ↪ handlesDocument(document))).forEachOrdered((document) →{
        docPath.add(document.getDocumentPath());
    });
    AWPrefs.setAllOpenedDocuments(docPath);
}
```

(b)