Adoption of Lambda-Expressions in Object-Oriented Programs does not necessarily decrease Source Code Size

[Draft Paper]

Sebastian Nielebock, Frank Ortmeier
Otto-von-Guericke Unviersity Magdeburg
Faculty of Computer Science
Institute of Intelligent Cooperating Systems
Chair of Software Engineering
{sebastian.nielebock,frank.ortmeier}@ovgu.de

ABSTRACT

Lambda-expressions are state-of-the-art in the functional programming paradigm, e.g. to easily handle functional behavior as a parameter or to temporarily implement a functionality, where it is actually needed.

These benefits motivate the developers of object oriented programming languages to extend their language by Lambda-expressions. When doing this, one out of several claims is that Lambda-expression are capable to decrease the source code size. Up to now, only little validation of this claim was conducted.

Thus, within this paper, we analyze, whether this claim is valid or not. For this purpose, we investigated the adoption of Lambda-expression and their effect on the source code size based on 110 open source projects, written in one of the three commonly applied object oriented programming languages C#, C++ and Java. Our results, obtained so far, seem to contradict this common claim. Moreover, the results indicate the opposite: the addition of Lambda-expressions is correlated with a more increased source code size than usual. Thus, we conclude that Lambda-expression does not necessarily decrease source code size.

Within this paper, we discuss potential reasons for this result, whereas a valid explanation is still an open research question.

Keywords

Lambda-Expression,Adoption, Object-Oriented Programming Languages,Source Code Size

1. MOTIVATION AND RESEARCH QUESTION

Lambda-expressions represent a key element of the functional programming paradigm. They allow to abstract and handle behavior as anonymous functions in the same way as object-oriented languages abstract and handle data and their related behavior as objects. Lambda-expressions forces programmers to describe small and stateless functionalities, where they are actually needed. By combining Lambda-expressions with complex data types such as collections, one can easily describe functions to be mapped on the collected data, filter data from that collection etc. Moreover, an extension of Lambda-expression - so-called closures - allow a closed context, e.g. surrounding variables, for Lambda-expressions to be used within the function's body.

These benefits motivate the developers of object-oriented (OO) programming languages,i.e. C#, C++ and Java, to incorporate Lambda-expressions into their syntax. With the adoption of Lambda-expressions, several claims about their influence on the source code were made. Here, four often proposed claims on Lambda-expressions[6, 5] say that

- they decrease the resulting source code size,
- they are more applicable in concurrent contexts, because of their statelessness,
- they allow easy composition of software functionalities,
- they have no serious impact on the software's performance.

In the past, these claims were investigated partially. For instance, when introducing Lambda-expressions in C++[6], the authors exemplarily showed the reduction of lines of code (LOC) and the constant performance of Lambda-expressions in C++. In particular they demonstrated on two existing software projects that the lines of codes decreases in comparison of the prior pure object-oriented solutions. In a study of different programming paradigms, the authors find out that the solutions of programming tasks solved with a functional language are significantly shorter than those of other paradigms, i.e. OO ones[14]. Based on these findings, we might assume that functional elements, especially Lamda-expressions, shorten the source code size in object oriented programming languages as well.

Gyori et.al. reasoned that Lambda-expressions are more applicable in concurrent contexts on the fact that several parallel libraries,namely Task Parallel Library, Parallel LINQ and Intel® Threading Building Blocks, have applied them more frequently. Within their paper, they also present a

© 2016 ACM. ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

refactoring tool, which automatically detect and insert Lambda-expressions. At least for the insertion of lambda-expression for Java's anonymous inner classes, they show the reduction of source lines of code (SLOC)[5].

Nevertheless, to the best of our knowledge, no one has studied the effects of the adoption of Lambda-expression, when programmers added them manually. That means, even though automatically refactoring of Lambda-expressions can reduce the source code size, this does not necessarily imply that programmers can reduce the code size by applying Lambda-expressions as well.

Therefore, we will study, if the adoption of Lambda-expressions by programmers of open source projects reduces the code size, i.e. source lines of code (SLOC) in the effected code base.

Thus, our paper focusses on the first of the four above mentioned claims therefore omitting the analysis of the three others. The second claim, which states that Lambda-expressions are more applicable in concurrent contexts, is planned to be investigated in a revised version of this draft paper. The remaining two claims are difficult to measure for real source code. Here, a meaningful metric for easy composition of Lambda-expression does not exist to the best of our knowledge. Concerning the performance it is hard to measure the effect of Lambda-expressions. First, because existing open source projects typically have a huge number of execution traces, one can hardly decide which traces to test. For example, assume that an inserted Lambda-expression slows down one trace A and speed up a trace B. If A was a more frequently used trace than B in program execution, the insertion would have a negative effect on the overall project. In opposition, the insertion would be benficial for the project, if B was used more frequently. Consequently, we cannot decide, whether an insertion has a positive or negative effect. Second, a changed performance might have a reason outside from the file/location where the Lambda-expression is inserted, due to side effects.

This study is comparable with previous work on the adoption of generics [10, 11, 7, 2, 3] for Java and C#. Especially, the differences in the adoption of generics in C# and Java[7] motivate us to analyze the adoption of Lambda-expressions for different programming languages.

Here, we only consider the commonly used OO programming languages C++, C# and Java. As recent rankings[1] show, these three languages are always in the top six of the most-applied languages. Even though all three languages add Lambda-expressions in the course of their development, they have been available for varying periods of time. Whereas C# has Lambda-expressions since version 3.0 in 2007, C++ added Lambda-expressions and closures in 2011 with version C++11 and even later Java followed C++ by adding them with Java 8 in 2014. Thus, we might conclude that C#-programmers have used Lambda-expressions more frequently than C++- or Java-programmers.

To answer our research question, we analyzed a total number of 110 open source projects from GitHub[2]. These represent well established projects as we discuss in section 2.2. The reason why we concentrate on these projects lies in the observation that Lambda-expression are hard to adopt correctly for freshmen and unexperienced programmers[14].

Concerning our research question, we considered the effect on the SLOC by adding a Lambda-expression. For this purpose, we analyzed the source code differences of every commit in the project's repository and examined if these differences contain a Lambda-expression. When a Lambda-expression was added in a code-difference, we examined if the related SLOC increased, decreased or stayed the same. We describe the methodologies and metrics in more detail in section 2.3.

Before presenting our experimental results, we perform two validity experiments (section 3.1). These experiments allow us to check the validity of the methodology for correctly finding Lambda-Expressions. This way, we want to be sure that neither false-positives, i.e. detected expressions are none, nor false negatives, i.e. existing expressions were not detected. Then we present the results for our research question in section 3.2 and summarize the main results in section 3.3. Although we validated the detection, we still see some potential threats to validity concerning our results, which we discuss in section 4.

Finally, we discuss the main results of our analysis and reveal consequences as well as further research directions (section 5).

2. ANALYZED DATA AND METHODOLOGY

In the following section, we give a short introduction into the syntax of Lambda-expressions for C#, C++ and Java. Readers who are already familiar with this syntax might want to skip this introduction. Further, we describe our selected experimental data, i.e. which projects we analyzed (section 2.2). In particular, we explain how we acquire data from GitHub projects and how we extract the added Lambda-expressions. Afterwards, we reason our methodology for finding Lambda-expressions (section 2.3).

2.1 Lambda-Expressions

```
(int x, int y) => x>y
(string s) => {
    Console.WriteLine("Wrote:"+s);
}
```

Listing 1: Example of Lambda-expression in C#

Our three investigated languages C#, C++ and Java have a slightly different syntax for Lambda-expression. Thus, we will shortly introduce these syntaxes, as they are important for the detection of Lambda-expressions in the source code. We describe them by depicting two example Lambda-expressions in each language to highlight the major differences. The first expression checks, if, for the input integer parameters x and y, the condition x>y holds. The second one prints a string variable s on the console.

As mentioned before, C# was the first of the three languages that introduced Lambda-expressions. Here, C# uses the "=>"-operator to map input parameters to an expression as it can be seen in Listing 1.

The Java-syntax is nearly the same except that the mapping-operator is depicted as "->" (see Listing 2).

In C++ the mapping operator "->" can be skipped. Thus, the Lambda-expression in C++ can be depicted as a usual method, whereas the method name is omitted (see Listing 3). Note, that the there exist mapping operator in C++, i.e. "‘->", too. Moreover, C++ allows closures that extend

[1]TIOBE Index January'16, RedMonk Programming Language Ranking January'16, PYPL PopularitY of Programming Language January'16
[2]https://github.com/

Lambda-expression to apply some kind of context by shifting surrounding variables or objects, especially this and super, via the bracket-syntax.

```
(int x, int y) -> x>y
(String s) -> {
    System.out.println("Wrote:"+s);
}
```

Listing 2: Example of Lambda-expression in Java

```
[](int x, int y) {
    return x>y;
}
[](string s) {
    cout << "Wrote:" << s << endl;
}
```

Listing 3: Example of Lambda-expression in C++

Note, that input parameters are not mandatory for any of the three Lambda-expression syntaxes. In this case the empty parameter list is depicted as two parentheses, i.e. "()". Furthermore, in all three languages Lambda-expression can be nested to the meaning that Lambda-expression can be placed within other Lambda-expressions.

After introducing the major syntactical differences of Lambda-expressions of the selected languages, we describe the criteria for selecting projects for the analysis.

2.2 Analyzed Projects

For the research question, we analyzed 110 projects from GitHub. As we discussed above, we focus only on well established projects with experienced programmers as they typically adopt Lambda-expressions correctly and efficiently[14]. Therefore, we filter the projects based on the GitHub REST API[3] with the following four criteria:

- the project has at least 200 files, as this guarantees a minimum level of source code size
- the project has at least 200 followers, as this guarantees that the source code is frequently maintained/tested
- the project exist at least six years, i.e. first commit before the 1'st January 2010, as this guarantees some level of maturity for this project
- the project's last commit is not older than the 1'st May 2016, as this guarantees that the project is still in the development process

By applying these criteria, we obtained *16* C#-projects, *31* C++-projects and *63* Java-projects. Note that, we actually obtained 65 Java-projects, one of which we had to omit due to non-accessible commits and another due to the overwhelming size of more than 100000 commits. Concerning the second project the analysis would currently last prohibitively longer. Nevertheless, we plan to include this project in a later version of this draft paper. Note, that all these projects are stored as Git-repositories. Therefore, we are able to store and analyze the code changes uniformly, as we describe in the next section.

[3]https://developer.github.com/v3/

2.3 Methodology

To analyze these projects on the influence of the adoption of Lambda-expressions on the code size, i.e. SLOC, we have to apply three basic steps:

1) locally store the code differences in a proper form for efficient analysis of the adoption of Lambda-expressions

2) find and store added Lambda-expression within these code differences

3) compare the effect on SLOC of code differences that added and didn't add Lambda-expressions, respectively

With respect to **the first step**, the general way to depict code differences is the textual form. Here, a commonly used tool is GNU diff[4], which detects textual differences - so called *hunks* - on the basis of the longest common subsequence[8, 15, 9].

Another method is to depict the source code as a (abstract) syntax tree (AST). Here, Lambda-expressions would be depicted as special nodes in the AST and thus the detection is less error prone than in the textual form. Unfortunately, the computation of tree differences, even using specialized, efficient algorithms, is currently at least ten times slower than calculating hunks[12, 4].

One possible solution might be, just to store the nodes of the AST and depict the differences between AST's as added and deleted nodes. Though faster than the tree comparison, this solution loses the context information, i.e. in which context the Lambda-expression is adopted. However, we need this information for later analysis, e.g., why some Lambda-expressions increase or decrease source code size, respectively.

Another technique is a declarative languages for Software Repository Mining, i.e. Boa, which allows to analyze data and meta-data of software software repositories from GitHub and SourceForge [1]. Even though it was quite successful applied in the analysis of Generics in Java-projects[2, 3], Boa's source code analysis is restricted to Java.

Thus, we stored the code differences between two different commits as hunks in a database. In particular, a Python-script examines the commits of all 110 projects, stored all changed, added and deleted source files, applies GNU Diff and stores the hunk in a SQLite database. This database also contains meta-data about the project, e.g. logged commit messages and timestamps of the commits.

Concerning **the second step**, we had to find Lambda-expressions in this hunks. Basically, when we want to automatically find Lambda-expressions or programming language features in general, we are confronted with the problem of deciding if the found features are correct and complete. The correctness describes the absence of false-positives, i.e. the absence of detected Lambda-expressions that actually are not any. The completeness expresses the absence of false-negatives, i.e. all existing Lambda-expressions in the source code are detected. Our method aims to find Lambda-expressions correctly and completely.

Since the detection of Lambda-expressions relies on textual differences, we used appropriate regular expression to

[4]https://www.gnu.org/software/diffutils/manual/diffutils.html

find Lambda-expressions. Here, we need to remove perturbing factors, i.e. comments or strings, because these could include the mapping operator, e.g. "=>", and therefore could lead to false positives. Thus, we removed comments and replaced strings in the source code before storing them in the database. Moreover, we manually checked the detected Lambda-expression (see validation experiments in section 3.1 for correctness and completeness.

The regular expressions have to detect multiple Lambda-expressions in a hunk so that the number of added Lambda-expressions can be correctly counted. Here, it is also important to correctly consider not added Lambda-expressions in a textual difference.

```
133c133,136
<       a = [](B*, const C&) -> Status {
    return OK(); };
---
>       a = [](const D&) {};
>       b =
>           [](B*, const D&, E)
>               -> OpTime { return OpTime();
      };
```

Listing 4: Example of an added and deleted Lambda-expressions as hunk that results in only one added Lambda-expression

For example the hunk in Listing 4 represents one deleted Lambda-expression (deleted lines are preceded by "<") and two added expressions (added lines are preceded by ">"). Thus, only one Lambda-expression was actually added, whereas another was only changed. Consequently the resulting number of added Lambda-expressions is one. Note that the resulting number might be zero or even negative, as the number of added and deleted Lambda-expressions stays the same or the number of deleted Lambda-expressions exceeds the added ones. As we are interested in the effect of added Lambda-expressions, we store this resulting number in the database as well.

The last step encompasses the analysis of the source code size. We measured source code size in source lines of code (SLOC), as lines of code are highly correlated with the effort of maintainability[13]. SLOC means that only lines with code are regarded, while omitting comments and blank lines. We already omitted comments in hunks to ensure correct detection of Lambda-expression and thus we additionally have to ignore blank lines when counting the SLOC.

To analyze the effect on SLOC, we distinguish two sub-questions with resepect to our main research question:

1) Does the SLOC increase, decrease or do not change within a hunk?

2) By how many SLOCs does the hunk increase or decrease?

The first sub-question describes a qualitative measurement, whereas the second one is a quantitative one. As the number of SLOCs is highly dependent on the project, we cannot compare the absolute number. Consequently, we measure the qualitative effect on the SLOC by the ratio:

$$effect_{qual} = \frac{hunk_{SLOC-inc} + hunk_{SLOC-tie}}{hunk_{SLOC-dec} + hunk_{SLOC-tie}}$$

where $hunk_{SLOC-inc}$ denotes the absolute number of hunks, whose SLOC increases, $hunk_{SLOC-dec}$ denotes the absolute number of hunks, whose SLOC decreases and $hunk_{SLOC-tie}$ denotes the absolute number of hunks, whose SLOC does not change (tied). We added the number hunks with "tied" SLOC two both decreased and increased hunks, as this guarantee to have a qualitative measurement even in cases where the addition of Lambda-expression has no effect on the SLOC. That means in cases, in which all hunks have the same size, i.e. $hunk_{SLOC-inc} = hunk_{SLOC-dec} = 0$, we still can calculate a ratio of 1.0, representing that source code size does not change. A value $effect_{qual} > 1$ means that the overall source code size increases, whereas a value $effect_{qual} < 1$ indicates a decreasing source code size. Note that case with solely increased hunks, we were not able to calculate a value for the qualitative effect.

Concerning the quantitative effect, we simply calculate the ratio of added and deleted SLOC:

$$effect_{quan} = \frac{SLOC_{add}}{SLOC_{del}}$$

Here, $SLOC_{add}$ and $SLOC_{del}$ denote the absolute number of added and deleted SLOCs, respectively. For example the source code in Listing 4 has the values $SLOC_{add} = 4$ and $SLOC_{del} = 1$ resulting in $effect_{quan} = 4$. By doing this, we are able to calculate the quantitative effect on single hunks. Similar to the qualitative case a value $effect_{quan} < 1$ denotes decreasing source code size, $effect_{quan} > 1$ denotes increasing source code size and $effect_{quan} = 1$ indicates that the source code size does not change.

To calculate these measurement, we examined the hunks in our database and stored the values of added and deleted SLOC for every hunk.

We compared the effect on SLOC on hunks that added Lambda-expressions, i.e. at least one additional Lambda-expression exist, and hunks that did not add Lambda-expressions, i.e. none additional Lambda-expression existed or Lambda-expressions were deleted. Thus, we could contrast the behavior of adoption of Lambda-expressions with usual case.

Based on our database, we were able to conduct experimental analysis to answer our research question by applying adequate SQL-commands. Before proceeding with the actual analysis, we describe how we validated the detection mechanism of Lambda-expressions for correctness and completeness to ensure corrects results for our analysis.

3. ANALYSIS OF THE RESEARCH QUESTION

The following section describes the results from our analysis up to this point. In the first sub-section, we describe how we validate the detection of Lambda-expressions, as the detection via regular expressions on hunks is threatened to be more error prone compared to other approaches.

In the section 3.2, we analyzed the effects of added Lambda-expression on the source code size, i.e. the SLOC. Finally, we discuss the consequences of our analysis' results in section 3.3.

3.1 Validation

As described in the previous section, deleting comments and replacing strings does not guarantee that the detection of Lambda-expressions is correct and complete. For this purpose, we conduct two validation experiments to reduce the threats of falsely detected (incorrect) or missed (incomplete)

Lambda-expressions.

For ensuring correctness, i.e. the absence of false-positives, we manually checked the added Lambda-expressions. In particular, we randomly chose different hunks with detected Lambda-expression and decided, whether, the hunks really contained a Lambda-expression or not, and if the detected number of Lambda-expressions were correct. In case of a false-positive, we revised the regular expression to correctly handle these edge cases. One of these example cases was for found in the C++ code shown in Listing 5. Here, an operator-definition was incorrectly interpreted as a Lambda-expression.

```
[]{
  // Lambda-expression with no parameters
}

operator[]{
  // re-defining []-operator
}
```

Listing 5: Example of a false positve Lambda-expression in C++

Further, we analyzed the completeness by running the regular expression on samples from the three different programming languages that contain Lambda-expressions in different variations. We also added the samples that we learned from the previous validation step.

Note that these two steps prove neither the completeness nor correctness. Nevertheless, by conducting this validation, we are more convinced that the detection is valid in the sense of completeness and correctness.

3.2 Analysis of the Effect on Source Code Size

By validating correctness and completeness of the detection process, we were able to get valid results from our database. In particular, we considered the qualitative and quantitative effects on SLOC of hunks that added or did not add Lambda-expressions as described in section 2.3.

First, we compared the effects on the level of hunks. That means, for the computation of the effect values, we only take those hunks into account, which have a certain property, i.e. added Lambda-expressions, or not. This denotes the "local" effect on SLOC, as this considers only directly affected hunks.

It is possible that hunks contain solely added or deleted lines. Here, calculating the effect values was meaningless, as it was clear that a hunk with solely added/deleted SLOCs had an increasing/decreasing effect. Moreover, solely added hunks usually describe an added feature or functionality in the source code, resulting in a high number of increased SLOCs. This would have influenced the effect values negatively, as for example a Lambda-expression that was added in the context of a new functionality, was never intended decrease source code size and thus was not capable of doing that. Consequently, we only considered those hunks that changed the source code, meaning that the hunk had at least one added and one deleted SLOC.

By filtering for such hunks, we noticed that only in 9 out of 31 C++-projects and in 7 out of 63 Java-projects such hunks with added Lambda-expression existed. This indicates that Lambda-expressions in these programming-languages are not as popular as for C#-projects, where all 16 projects contained such hunks. Projects that had not added Lambda-expressions at all, or whose effect values could not be calculated, e.g. for no hunk the SLOC decreased or did not change, were excluded from the analysis of added Lambda-expressions. Thus, in our analysis "hunks that added Lambda-expressions" only referred to such hunks from those projects, that added Lambda-expressions and whose effect value could be calculated.

The resulting confidence intervals, around the mean of the effect values over the number of projects, is depicted in Figure 1. The general trend for the qualitative and the quantitative effects tends to increase SLOC both hunks with added and non-added Lambda-expressions, as the mean values of $effect_{qual}$ and $effect_{quan}$ is greater than one. The only exception for this trend is the mean value of the qualitative effect of hunks that added Lambda-expressions in Java-projects in Figure 1e.

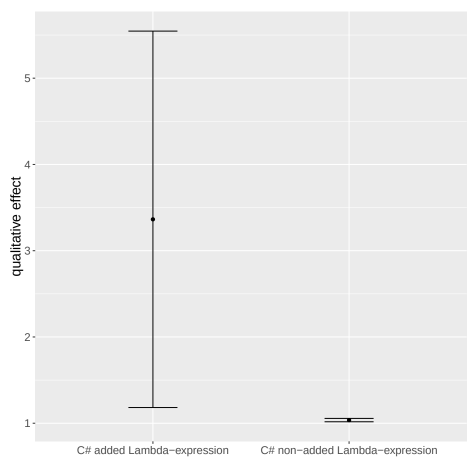
The confidence intervals for C++ and C#-projects indicate that hunks that added a Lambda-expression have a significantly higher qualitative effect value than hunks that did not add a Lambda-expression. Moreover, for C# the quantitative effect value shows the same behavior. For all other intervals we cannot conclude a significant difference between the effect values, i.e. Java's effect values and C#'s quantitative effect value. Thus, we conclude that our data does not indicate that Lambda-expressions significantly decrease source code size, but rather the opposite. This is quite surprising, as it contradicts the common claims about Lambda-expressions.

As Lambda-expression might have a global effect instead of a purely local one, we also considered the effect values on the level of files. This means, we compared the effect values of all hunks in a file that added Lambda-expressions with those that did not add a Lambda-expression. Note that we deliberately ignored the effect values of all hunks within a commit. Here, we did not think that these effect values were meaningful, because aside from the added Lambda-expressions, there were so many other factors that could have influenced the SLOC in a complete commit.

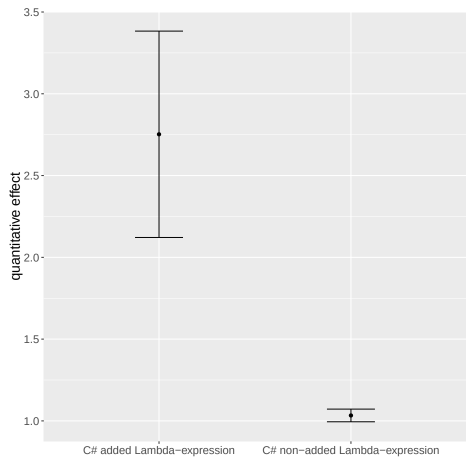
When considering the effect on a file, we had to include hunks that solely added or deleted lines. For example a Lambda-expression could have increased the SLOC in one hunk of a file, while it had decreased in another. The resulting effect values represent a more "global" effect on the source code size, i.e. the effect on the file size. Nevertheless, the results stay nearly the same, as the confidence intervals in Figure 2 express. In particular, both effect values of the C#-projects for hunks with added Lambda-expressions are significantly higher than those of hunks that did not add Lambda-expressions. Concerning the effect values of the other programming languages, only the quantitative effect value of the Java-projects behaves similarly.

Both the effect values on local hunks and those on the files indicate that the SLOC more increases, when a Lambda-expression is added, compared to the non-added case. As these results seem contradict the common claim about Lambda-expression, we further analyzed the hunks to find reasons for this behavior.

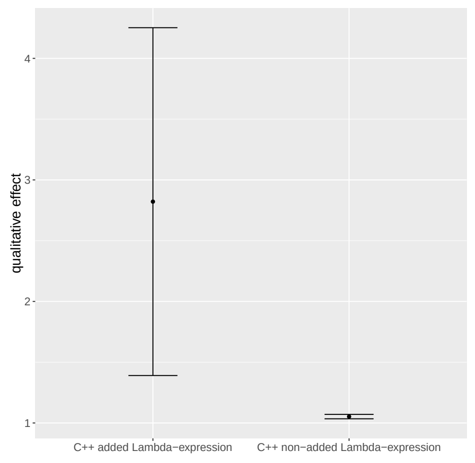
For this purpose, we analyzed the logged commit message, to figure out, for which tasks programmers commonly added Lambda-expressions to the source code. Moreover, we considered, whether some of these tasks were correlated with the increase or decrease of the source code size. Such a correlation could indicate, whether for example a Lambda-expression is more suitable to decrease source code size in



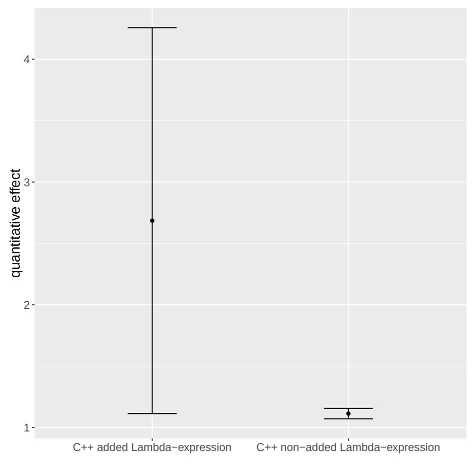
(a) Qualitative effect on SLOC for local hunks in single C#-projects



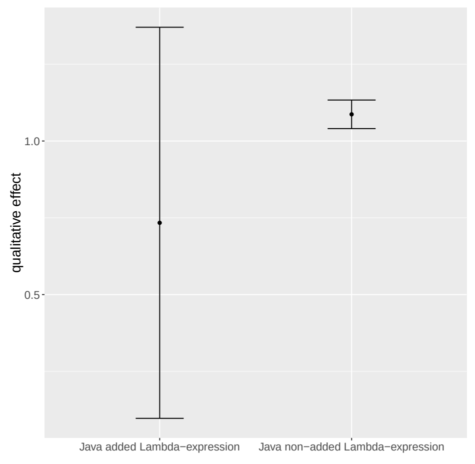
(b) Quantitative effect on SLOC for local hunks in single C#-projects



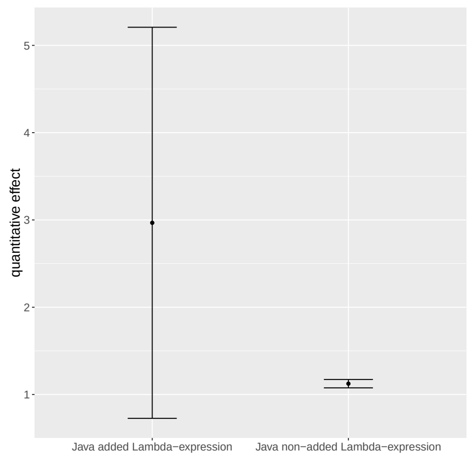
(c) Qualitative effect on SLOC for local hunks in single C++-projects



(d) Quantitative effect on SLOC for local hunks in single C++-projects

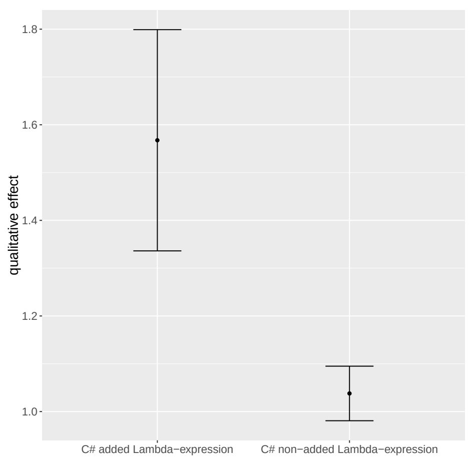


(e) Qualitative effect on SLOC for local hunks in single Java-projects

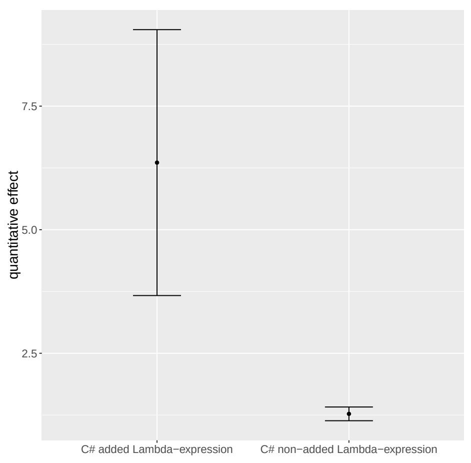


(f) Quantitative effect on SLOC for local hunks in single Java-projects

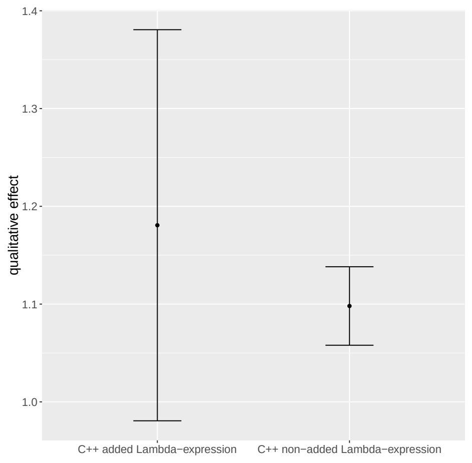
Figure 1: Comparison of the confidence intervals with $\alpha = 0.9$ of the qualitative and quantitative effect values between hunks with added and non-added Lambda-expressions



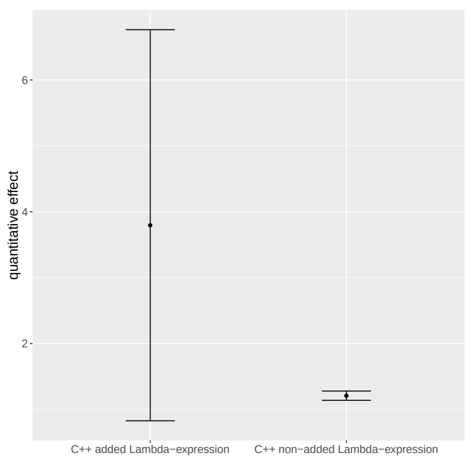
(a) Qualitative effect on SLOC for all hunks in changed file in single C#-projects



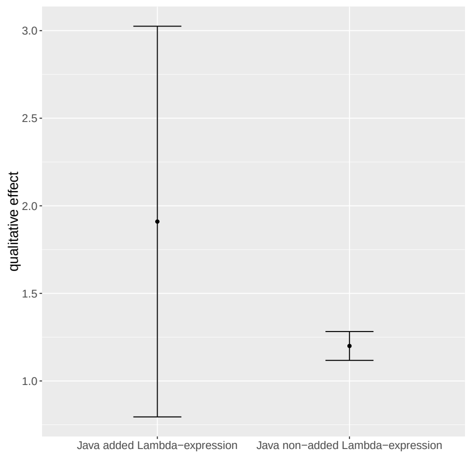
(b) Quantitative effect on SLOC for all hunks in changed file in single C#-projects



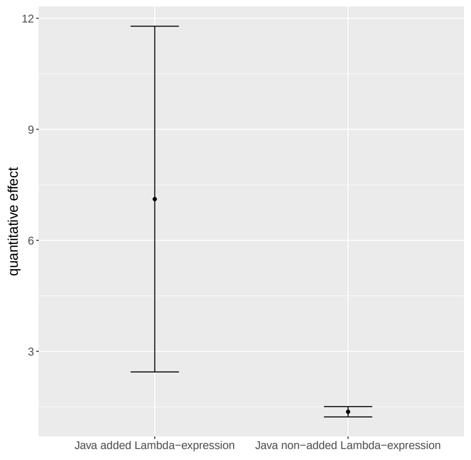
(c) Qualitative effect on SLOC for local hunks in single C++-projects



(d) Quantitative effect on SLOC for all hunks in changed file in single C++-projects



(e) Qualitative effect on SLOC for all hunks in changed file in single Java-projects



(f) Quantitative effect on SLOC for all hunks in changed file in single Java-projects

Figure 2: Comparison of the confidence intervals with $\alpha = 0.9$ of the qualitative and quantitative effect values between all hunks in a file with added and non-added Lambda-expressions

particular tasks.

To infer the task, we applied a word frequency analysis to commit messages using the appropriate R tm-library, which enabled us to directly apply common analysis methods. Here, we collected commit messages in two ways:

1. hunk-collection: for every hunk that has a particular property, e.g. added a Lambda-expression, it's corresponding commit message is taken into account for frequency analysis

2. commit-collection: for every commit the corresponding commit message is taken into account at most once, if at least one hunk has a particular property, e.g. an added Lambda-expression

The first collection represents "weighted" commit messages with respect to the hunks. Thus, a logged commit message that added multiple Lambda-expressions in different hunks is replicated more often. In opposition to that the second collection is more robust with respect to outliers, i.e. a commit that heavily added Lambda-expressions and therefore biases the overall frequency.

Furthermore, we clustered particular words to accumulate the occurring frequencies. This was necessary, as word frequency usually results in small percentages,i.e. $<= 2\%$, whereas multiple words could be replaced by a synonym. By manually examining the frequent words for programming tasks, we could infer the following five tasks from the commit messages:

- refactor - refactoring the source code
- add - added functionality in the source code
- delete - deleted functionality in the source code
- test - created test cases for the source code
- fix - fixed errors in source code

The results of the word frequency analysis are restricted to those five tasks, as, even though removing common and most perturbing words, some meaningless words still remain in the results. Thus, the results can be interpreted as frequent tasks.

As stated before, the number of C++- and Java-projects that actually applied Lambda-expressions was quite small and therefore the same is true for the number of related commit messages. Moreover, only C#-projects consistently showed the potential increasing effect in both effect values. That is the reason, why we only present the results of the frequency analysis for C#-projects.

The overall trend indicates that the most frequently applied task is the refactoring one. With respect to the task frequency based on the hunk-collection, refactoring occurs more frequently, when a Lambda-expressions are added and the source code size decrease or does not change at all. In particular, when adding a Lambda-expression as depicted in Figure 3e, "refactor" is nearly twice as frequent as in the corresponding non-added case in Figure 4e. Another interesting observation in the hunk-collection is that an "adding"-task occurs also more frequently, when SLOC is decreasing or is not changing, when Lambda-expressions are added.

One possible interpretation could be that, if Lambda-expression are applied during the task of refactoring or adding functionality, the adoption of the Lambda-expression is more capable to reduce source code size. Nevertheless, a preliminary experiment showed the opposite to be true. In particular, we analyzed the effect values of all hunks, whose related commit message contained at least one word that referred to a refactoring task. Here, the effect values of hunks that added Lambda-expressions remain larger than those of the hunks that did not add Lambda-expressions.

Referring to the frequency, based on the commit-collection between hunks, we cannot infer any major difference between hunks that add or do not add Lambda-expressions. Consequently, we cannot infer relations between task types and the adoption of Lambda-expression as well.

In the following we sum up the main results from the previously described analysis.

3.3 Summary of the Main Results

Taking all results into consideration, we come up with the statement that the adoption Lambda-expressions do not necessarily decrease source code size. In particular, our experiments show that the qualitative and quantitative effect is significantly higher at least for the investigated C#-projects. With respect to projects, written in other programming languages, i.e. C++ and Java, the number of hunks that added Lambda-expressions is quite small. Thus, this increasing effect cannot be significantly inferred for the other programming languages. Nevertheless, no result indicates a significant different behavior, i.e. decreasing SLOC based on an addition of a Lambda-expression.

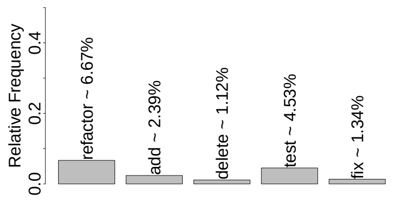
Up to now, we were not able to find a plausible explanation for the increasing effect on the SLOC of Lambda-expressions. An attempt to analyze frequent tasks, inferred from the logged commit messages, lead to the observation that the tasks "refactor" and "add functionality" are more frequent when Lambda-expressions were added and SLOC decreased. Nevertheless, a preliminary experiment was not able to confirm that, within a refactoring task, an added Lambda-expression decrease SLOC.

In the next section we discuss, some factors that could harm the validity of our results. This is important also for further analyses, which we plan to conduct.

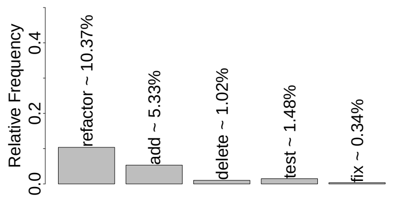
4. THREATS TO VALIDITY

Our analysis might have potential threats that we consider in this section.

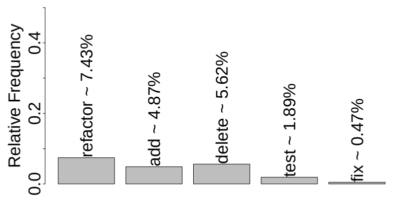
One major point of criticism is that even though we removed strings, blank lines and comments, we did not consider the subjective coding styles of programmers. Code style could lead to increased SLOC. For example some styles prefer to add an additional line for the brace, indicating the beginning of a function block, whereas others prefer to keep the brace in the line of the function declaration. Another style restricts the programmer to break up lines longer than 80 characters, leading to more SLOC. One idea of how to overcome this problem is to count number of statements instead of SLOC. However, hunks can represent partially incomplete extracts of the source code, on which counting statements is not always possible. Another idea could be to count characters instead of SLOC. This approach could lack robustness concerning the length of variable and method names. Finally, we could also try to normalize the investigated source code with respect to a particular style. However, this is hard to perform on the hunks, as some SLOC are not complete and therefore cannot be normalized, e.g. the



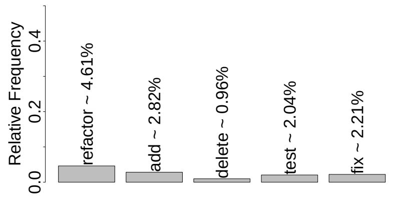
(a) Frequent tasks of increased hunks based on hunk-collection



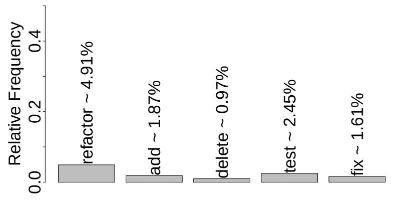
(b) Frequent tasks of decreased hunks based on hunk-collection



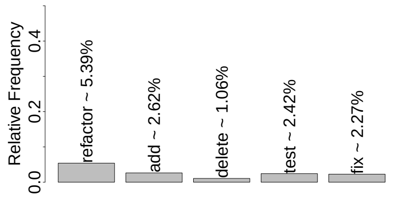
(c) Frequent tasks of not changed hunks based on hunk-collection



(d) Frequent tasks of increased hunks based on commit-collection

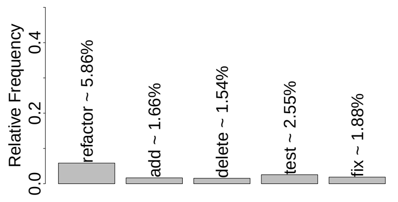


(e) Frequent tasks of decreased hunks based on commit-collection

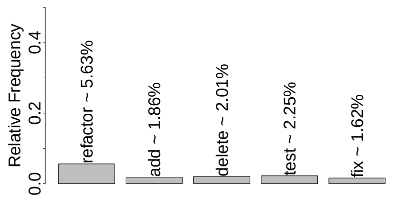


(f) Frequent tasks of not changed hunks based on commit-collection

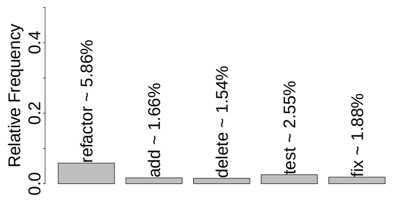
Figure 3: Frequent tasks of hunks that added Lambda-expressions



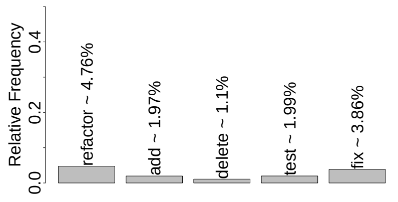
(a) Frequent tasks of increased hunks based on hunk-collection



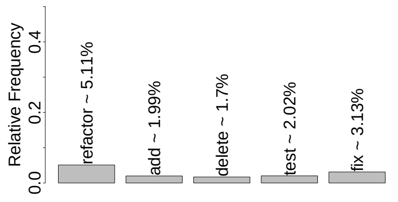
(b) Frequent tasks of decreased hunks based on hunk-collection



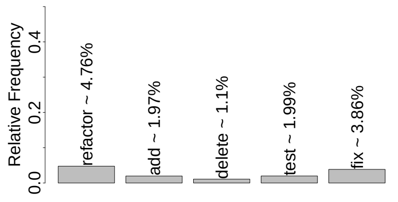
(c) Frequent tasks of not changed hunks based on hunk-collection



(d) Frequent tasks of increased hunks based on commit-collection



(e) Frequent tasks of decreased hunks based on commit-collection



(f) Frequent tasks of not changed hunks based on commit-collection

Figure 4: Frequent tasks of hunks that do not added Lambda-expressions

normalization of a single added line differs, if it was added in another context.

When we analyzed added and deleted SLOC, we did not consider whether these SLOC were somehow related to a Lambda-expression. Accordingly, we cannot be sure, if the increase of the source code is really caused by the Lambda-expression, or is, in some sense, “noise”. Even though we diminish this “noise” by comparing those hunks that added and those that did not add Lambda-expressions, we plan to manually examine the hunks for this purpose.

Another threat is the data itself. This data represents only established open-source-projects and thus the statements we deduced, do not necessarily apply to other projects. Additionally, only a small number of projects from C++ and Java already applied Lambda-expressions. Thus, at least for these programming languages the truth of our hypothesis is uncertain. To overcome these threats, we need to include more projects in the data to investigate, e.g. by softening the criteria for project acquisition. This might also help to include projects that seem to be well established but do not follow our formal criteria up to now.

Moreover, the analysis does not represent a guided experiment. This means that programmers, who applied Lambda-expressions, were not necessarily concerned with the task to decrease source code size. Consequently, we cannot be sure, if the increasing behavior is caused by the Lambda-expressions or by some unknown, hidden factor. Because of this, some guided experiments have to be conducted to confirm our hypothesis that Lambda-expression do increase source code size.

Referring to this previous threat, it might be that even though the increase of SLOC is greater with Lambda-expressions, this does not mean that this increase has negative consequences for the source code at all. For example, the more fluent style of Lambda-expression can be more natural for programmers and thus easier to understand, even though longer in terms of SLOC. In addition, some Lambda-expressions could follow task other than reducing source code size, i.e. applied in a concurrent context.

Taking these threats into account, we discuss the main results of this paper and some future directions in the following, final section.

5. CONCLUSION AND FURTHER WORK

In this paper we investigated the claim that Lambda-expressions usually decrease source code size in object oriented programming languages. For this purpose, we acquired source code changes from 110 open source projects written in one of the three commonly applied object programming languages C#, C++ and Java. We defined measures for the effect on the source code size, i.e. source lines of code, and analyzed the effect values locally, on code differences, and globally, on source code files.

Our results, as obtained so far, appear to contradict the common claim that Lambda-expressions decrease source code size. The opposite may even be the case. Lambda-expression seem to correlate with larger increases in source code size than in the case of no Lambda-expressions. Up to now, the reasons for this behavior remains unclear.

Consequently, in the ongoing research, we intend to obtain typical use cases, in which Lambda-expressions are applied. Based on these use cases, we will be able to decide, whether some use cases are more capable of decreasing source code size than others.

We can also result in some design patterns as well as anti-pattern for the use of Lambda-expressions. This might also extend the work of previously developed, automatic refactorings[5].

In addition we are going to consider other claims, i.e. if Lambda-expressions are usually applied in concurrent contexts.

6. REFERENCES

[1] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 422–431. IEEE Press, 2013.

[2] R. Dyer, H. Rajan, H. A. Nguyen, and T. N. Nguyen. A large-scale empirical study of java language feature usage. 2013.

[3] R. Dyer, H. Rajan, H. A. Nguyen, and T. N. Nguyen. Mining billions of ast nodes to study actual and potential usage of java language features. In *Proceedings of the 36th International Conference on Software Engineering*, pages 779–790. ACM, 2014.

[4] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE ’14, pages 313–324, New York, NY, USA, 2014. ACM.

[5] A. Gyori, L. Franklin, D. Dig, and J. Lahoda. Crossing the gap from imperative to functional programming through refactoring. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 543–553. ACM, 2013.

[6] J. Järvi and J. Freeman. C++ lambda expressions and closures. *Science of Computer Programming*, 75(9):762–772, 2010.

[7] D. Kim, E. Murphy-Hill, C. Parnin, C. Bird, and R. Garcia. The reaction of open-source projects to new language features: An empirical study of c# generics. *Journal of Object Technology*, 12(4), 2013.

[8] W. Miller and E. W. Myers. A file comparison program. *Software: Practice and Experience*, 15(11):1025–1040, 1985.

[9] E. W. Myers. An o(nd) difference algorithm and its variations. *Algorithmica*, 1(1-4):251–266, 1986.

[10] C. Parnin, C. Bird, and E. Murphy-Hill. Java generics adoption: how new features are introduced, championed, or ignored. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, pages 3–12. ACM, 2011.

[11] C. Parnin, C. Bird, and E. Murphy-Hill. Adoption and use of java generics. *Empirical Software Engineering*, 18(6):1047–1089, 2013.

[12] M. Pawlik and N. Augsten. Rted: a robust algorithm for the tree edit distance. *Proceedings of the VLDB Endowment*, 5(4):334–345, 2011.

[13] D. I. Sjøberg, B. Anda, and A. Mockus. Questioning software maintenance metrics: a comparative case study. In *Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement*, pages 107–110. ACM, 2012.

[14] P. M. Uesbeck, A. Stefik, S. Hanenberg, J. Pedersen, and P. Daleiden. An empirical study on the impact of c++ lambdas and programmer experience. In *Proceedings of the 38th International Conference on Software Engineering*, pages 760–771. ACM, 2016.

[15] E. Ukkonen. Algorithms for approximate string matching. *Information and control*, 64(1-3):100–118, 1985.