Analysing Lambda Usage in the C++ Open Source Community

Jonathan Bengtsson, Heidi Hokka
Abstract - Object-oriented languages have made a shift towards incorporating functional concepts such as lambdas. Lambdas are anonymous functions that can be used within the scope of other functions. In C++ lambdas are considered difficult to use for inexperienced developers. This implies that there may be problems with lambdas in C++. However, studies about lambdas in C++ repositories are scarce, compared to other object-oriented languages such as Java. This study aims to address a knowledge gap regarding how lambdas are used by developers in C++ repositories. Furthermore, examine how developer experience and software engineering practices, such as unit testing and in-code documentation, correlates with the inclusion of lambdas. To achieve this we create a set of tools that statically analyse repositories to gather results. This study gained insight into the number of repositories utilising lambdas, their usage areas, and documentation but also how these findings compare to similar studies’ results in Java. Further, it is shown that unit testing and developer experience correlates with the usage of lambdas.

Index Terms - C++, Lambda Functions, Static Code Analysis, Mining Repositories, Software Engineering Practices

I. INTRODUCTION

C++ and Java support object-oriented design by their inclusion of objects, classes and inheritance [1]. Both languages have been in the top 10 most used languages on GitHub along with other object-oriented languages such as Python, JavaScript, PHP and C# [2]. It may indicate that languages supporting an object-oriented paradigm are dominant among open-source developers. Despite their popularity, object-oriented languages have continued to evolve and make a shift to accommodate the functional programming paradigm. For instance, functional concepts have been introduced into C++ and Java in the form of lambda functions [3], [4]. Lambdas in C++ are anonymous function objects that can be used to encapsulate code for later invocation [5]. A lambda function is used for short snippets of code that may be used for encapsulating behaviour that is only needed once or instead of iterators. However, researchers have shown that lesser experienced developers are slower or unable to solve tasks in C++ with lambdas when compared to iterators [6]. This implies that there may be problems connected to lambda usage in C++, which compared to Java has not been studied extensively.

Lambdas in Java repositories have previously been analysed with JavaParser to gain insight into where the feature has been used by developers [7]. Further, it has been measured how often the lambda expressions are accompanied by a comment in Java repositories [8]. The existence of lambdas in concurrent code and general use cases has been coarsely investigated for C#, C++ and Java repositories [9]. However, a less coarse and similar systematic analysis on lambda usage in C++ repositories has not been conducted, to the best of our knowledge.

Therefore, an empirical analysis of lambda usage in C++ may provide insight into how developers utilise this feature and unearth new problem areas worth pursuing. The purpose of this study is to address this problem by statically analysing C++ repositories for their inclusion of lambdas and how it may differ against previous results in Java. Additionally, if it’s possible to discern a link between usage of lambdas, developer experience and the presence of software engineering practices.

II. PROBLEM STATEMENT

In a controlled experiment it was shown that usage of C++ lambdas was dependent on developer experience. Experienced developers could solve the tasks with either lambdas or iterators, but lesser experienced developers had difficulties solving the problems with lambdas [6]. Other researchers found that many of their participants were unable to adequately document lambda expressions [8]. These insights imply that there are problems hindering the usage of lambdas. However, consequences when utilising lambdas have not been quantitatively measured with C++ repositories. The purpose is to gather empirical knowledge about lambda usage in C++ and how it may be affected by developer experience, and in turn affect the presence of different software engineering practices, specifically in-code documentation and unit testing. The results will then be able to be compared to previous results in Java. Additionally, it will be further investigated whether these differences in
utilisation and documentation can be measured by doing static code analysis.

III. RESEARCH QUESTIONS

Lambdas were integrated in C++ (version 11) [3] in 2011 and for Java (version 8) in 2014 [4], [10]. Both languages have been compared for their suitability under specific conditions, such as performance in numerical computing, program speed and ease of programming [11]–[13]. While both languages have been separately measured for their feature usage, cross-language feature acquisition has not been compared. Comparing the usage of similar features between both languages may result in objective data depicting which language has succeeded better in getting developers to acquire new features of a language. This could open up new questions regarding why and how a language could be modified to prevent underuse of implemented features. To solve the unavailability of information about lambda usage in C++ repositories, a number of research questions have been formulated. The result of these research questions will then be compared with similar studies made in Java.

RQ1. How does the usage of lambda-functional programming in open source software in C++ differ from Java?
RQ2. How does the usage of lambda-functional programming affect in-code comments in open source software in C++ compared to Java?

It is suggested that successful lambda usage was dependent on developer experience [6], however it is worth investigating if this relationship is valid in a subset of C++ repositories on GitHub. Since GitHub contains developers of different experience levels, a hypothesis may be that some form of correlation of C++ lambda usage and developer experience exists. This may result in that this feature is possibly underused by less experienced developers on GitHub. However, it may also be of interest to investigate how lambdas, in return, may affect the inclusion of unit testing, which Mazinanian et al. considered to be evidence of quality when evaluating repositories [7].

Since lambda functions are anonymous, they may pose a challenge in unit testing, where they can either be tested on their behaviour or by replacing them with method references [14], [15]. Furthermore, testing of lambdas can be unwarranted due to their simplicity or reliance on other functions [14]. However, [7] concluded that lambdas existed more often in test code than in production code. Since this can be contradictory and problematic, does it make it less or more likely for C++ repositories to include testing if they are using lambdas?

RQ3. How does the developer experience correlate with the choice to include lambda-functional programming in C++?
RQ4. How does unit testing differ between C++ repositories which are either utilising or not utilising lambda-functional programming?

IV. LIMITATIONS

Only C++ source and header files in master branches of repositories will be analysed for containing lambdas. Forks of the same project will not be included. Since the code will be analysed statically, the code in the repository does not have to be runnable. The construction will not support the mass downloading of repositories. The tool will not be tested for compatibility on other systems than Windows. The performance of the tool developed will not be measured. However, correctness in labeling lambda expressions will be considered. The initial part of lambdas will be matched row by row with a regex, a lambda with the capture and the params on different lines will not be matched. A user must have made at least two commits with the commit interval being greater than a day to the same repository for developer experience to be calculated and considered. The measurement will be done based on the author’s name in the commit, however it will not be able to tell if the author is committing under a different or similar name, so these will be considered as separate users. Further we have decided to place some limitations on the breadth of our analysis. The categorisation of lambda usage will focus on the locations a lambda can be introduced and how it is used. The context or purpose of lambdas will not be considered. For comments, we will base our assumptions on the location of comments within or above lambdas, not whether or not it can be subjectively decided if the comment and lambda may be related.

V. BACKGROUND

In this section we describe the terms and features that are good to have knowledge of throughout the rest of the study.

1. Lambdas

Lambda functions, also called lambda expressions or lambdas, are anonymous functions that can be declared within the scope of other functions. They are alternatives to using free functions and function objects (functors) and were integrated into C++ (version 11) in 2011 [3]. A lambda function is a short snippet of code that is not worth naming, and will not be used again. The C++ concept of a lambda function originates in the lambda calculus and functional programming. In C++ a lambda expression consists of three main parts, captures, params and body [16].
2. **Differences between Java and C++**

Java has been influenced by C++, however the languages have some differences [1]. Java is platform independent due to its execution inside a virtual machine [17], which makes it possible to run Java programs on any machine that has a Java runtime implementation. Running the code in a virtual machine is considered safer and may result in smaller programs [31]. Java is also a memory-managed language, which means that memory allocation and freeing is automatic. In Java this means that the developer is unable to manipulate the memory directly [19]. C++ places the responsibility of handling memory on the developer in that each object allocated with new should also be deleted to not cause memory leaks [20]. Lambdas may have an effect on memory management if variables are captured. In Java, capturing variables in lambdas is done implicitly and by reference, which means that the lambda has access to variables outside of its own scope. However, the variables referenced inside the lambda are required to be final. In C++ the capturing has to be done explicitly by placing the variables in the capturing group of a lambda. The variables can be passed by value, by reference, or both if using multiple variables [9]. The variables passed to a C++ lambda may also be mutable [16]. Java further requires that lambda expressions are targeted towards a specific functional interface which decides the number of arguments and the type of the result, if any [21]. This means that you need to know which functional interface to use for your lambda before creating and assigning it to a variable for later use. In C++ you don’t have to use a functional interface and can instead store the lambda in an auto variable [16]. The type is deduced automatically when using auto [22].

3. **Static code analysis**

To examine code without executing the program is called static code analysis which is a method used in debugging computer programs. Static code analysis can provide information about potential faults, vulnerabilities or spelling errors in code. The process may also provide an understanding of the code structure and help ensure that the code adheres to coding standards and guidelines [23]. An advantage of using static code analysis is the superior speed compared to manually reviewing code. Another advantage is the accuracy, when manually reviewing code there might be human errors, but an automated tool does not make these errors. Some limitations with static code analysis are that the tool is as good as the rules they use to scan with and it may produce false positives and false negatives [24].

4. **Mining Repositories**

A repository is a type of project storage using version control. Version control makes it possible to keep a history of edits in a project’s structure and code. Updates to a project are done in the form of commits, which retain knowledge about what has changed since a previous update. This makes it easier for contributors to collaborate on the same project [25]. We use the commit history of repositories to calculate each author’s developer experience.

GitHub may be used to remotely store repositories that are using git as their version control system [26]. In 2018 GitHub surpassed 100 million repositories [27]. Herzig and Zeller were convinced that public repositories from GitHub could be data-mined to gain unbiased and initial empirical evidence of a problem, but that it could also contain a lot of noise [28]. Since it was difficult to know which repositories contained actual engineered software projects, Munaiah et al. used a machine learning classifier that had been trained on actual software engineering projects. They used the classifier on more than 1.8 million GitHub repositories which resulted in a dataset they named RepoReaper [29]. We used this dataset to filter engineered C++ software repositories for our analysis.

5. **Developer Experience**

Developers’ experience is a variable measurement used in studies to depict its correlation with some other aspect of software engineering such as commit bugginess [30], usage of asserts [31], secure usage of cryptography [32], ability in solving programming tasks [6], program comprehension [33] and etc. A developer’s experience has been measured by how far a student has gotten in their studies, by work experience in years for a professional [6] and by assigned staff levels in a company [34]. However, open source projects may lack this type of information about the developers or a project may not have a hierarchy of contributors if it is decentralised [35]. Therefore developer experience has been measured differently in open source communities. Measurements applied in open source communities including how well a developer may match a job description by analysing the READMEs of projects the developer has participated in [36], the number of commits made to a single method in the repository [31] and by how long a developer had been contributing to a specific project by the difference in days between their first and a later commit [30]. We adapt the final method from [30] to measure a developer’s experience by the time between their first and last commit.

6. **Related Work**

Static code analysis tools can be used to discover problems in code, such as errors, vulnerabilities and adherence to coding standards [23]. Zampetti et al. found that static code analysis tools contributed to the swift resolution of build breakages in continuous integration pipelines [23]. However, the usage of static analysis tools is not limited to the maintenance of a single project. The tools have also been used to analyse repositories in order to gain empirical insight into how features are used. Nielebock,
Heumüller and Ortmeler used static analysis tools to gain insight on lambda usage in concurrent code present in open source repositories written in C#, C++ and Java [9]. Mazinan et. al. analysed repositories to understand the reasons for why and how lambdas were used in Java [7]. Alqaimi, Thongtanumam and Treude [8] measured the existence of lambda expressions in Java repositories and how often the expressions were accompanied by a comment. Nielebock. Heumüller and Ortmeler covered lambda usage in C++ repositories by their frequency and their presence in files that included threading libraries [9]. Further they classified usage areas based on filenames in the projects using lambdas and discovered that files with a certain classification were more or less likely to use lambdas. The usage of lambdas has also been evaluated against developer experience.

Uesbeck et al. depicted that lesser experienced developers had difficulties solving problems using C++ lambdas in controlled tests [6]. Uesbeck et al. divided experience into students and professional developers, with students being divided into further groups depending on how far they were in their programming studies and developers by their self-reported years of experience [6]. But this is not the only way developer experience has previously been measured. Gill and Kemerer mentioned that a developer's experience could be measured by their work experience in years, but that it was problematic since it did not take into account any prior education of the developer. Instead they chose to use the staff levels of employees (junior engineer, engineer and senior engineer) as their indicator of developer experience [34]. However, open source projects may lack this type of hierarchy or centralisation of contributors [35]. Therefore developer experience may have to be measured differently in open source communities.

Eylwolf, Tan and Lam measured developer experience in repositories after how long a developer had been contributing to a specific project [30]. They compared the timestamps between two commits, first commit and one later in time, to get the developer’s experience in days within the project and compared it against the introduction of bugs. Casalnuovo et al. measured experience by the amount of commits made to a single method by a developer [31]. Hauff and Gousios attempted to match GitHub profiles to job advertisements with specific experience requirements by extracting concepts from READMEs in repositories that a developer had participated in [36]. The differing ways of measuring developer experience in open source communities may indicate that an agreed-on standard does not exist.

Developer experience, despite its numerous ways of being measured, has shown that experience may be linked to the ability to solve problems. Feigenspan et al. compared the answers to questionnaires about programming experience from students and their ability to solve ten programming-comprehension tasks [33]. The students’ own estimation of their programming experience had the highest correlation with the number of tasks solved. Wilcox and Lionelle showed that students with previous exposure to programming performed better in exams and quizzes in introductory programming courses than those who had no previous experience [37]. However, while lambda usage has been compared to developer experience in C++ during controlled forms with participants, these measurements have not been applied against repositories and their authors. We intend to try out a method for measuring the mean experience of a repository and compare it to the existence of lambdas. This study addresses the lack of knowledge about how C++ lambdas are used by also analysing where lambda expressions are used in C++ repositories. We also intend to compare our results against previous studies about Java lambda usage and comments to see if we can motivate that there is a difference. Further, using values from RepoReaper we will see if lambda usage has a correlation on the presence of unit testing code.

VII. RESEARCH METHODOLOGY

This study investigated the usage of lambdas by cloning 500 repositories and statically analysed their inclusion of lambdas and if the lambdas were accompanied with comments. Further, this study also compared both developer experience and unit test values for repositories utilising and not utilising lambda expressions. A summary of how this was done is presented in Table 7.1. and will be detailed in the upcoming sections.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collecting Data</td>
<td>Find and clone 500 unique software engineering repositories using C++ as their primary programming language.</td>
</tr>
<tr>
<td>Construction of Tools</td>
<td>Three different tools, the first finds all lambdas, the second measures the comments above and inside the lambdas, the third measures developer experience and gives the result as mean developer experience in days.</td>
</tr>
<tr>
<td>Categorise &amp; Analyse &amp; Measure</td>
<td>Categorise lambdas, analyse lambdas and their comments, measure development experience and unit test value.</td>
</tr>
</tbody>
</table>
1. **Collecting Data**

The RepoReaper dataset was used to find repositories that were considered engineered software projects on GitHub [29]. The repositories may be more relevant for what we wanted to measure and compare to another study using the same dataset. The dataset was queried by selecting all rows which had the language C++ and where the randomforest.org value was set to 1, which meant that the repository was classified as an engineered software project with the random forest classifier in RepoReaper. The filtered dataset was split into files of 100 owner and repository names each, which would be used to clone the repositories in batches. The output from filtering was not sorted and remained the same ordering as in the dataset. The repositories were manually cloned with a one-liner Fig 7.1 which iterated over the repository names present in the file. The cloning continued until we had successfully cloned a total of 500 repositories which we could use in our analysis. We also extracted the unit_test values for the analysed repositories from the RepoReaper dataset. All data from RepoReaper was saved into a database where information about the repository name, the owner and the unit test could be found.

```bash
for f in `cat <filename>.csv`; do `git clone https://github.com/$f`; done
```

Fig 7.1. The one-liner used to clone the repositories from a list of repository owners and names.

2. **Construction of Tools**

This section details the construction of the tools used to get the results. We constructed three tools in total that each had their own role in gathering and presenting the results.

2.1. **Lambda Detector**

When all repositories were cloned we could start checking if a repository contained a lambda. To do this we created a lambda detecting tool. The tool is searching all folders in every repository for a file-extension that is used for C++, such as .cpp or .hpp. The tool then examines the C++-file line by line to check if there is a lambda. A lambda can be constructed in different ways with different characters inside the captures. To be able to get all different types of lambda constructions a regular expression (regex) was used. Regexes can be used to pattern match strings. With regex it was possible to capture all characters inside the captures that a lambda could use. We used two different regexes, one for the lambda detector and one to prevent matching code that looks like a lambda but is not. When developers comment code, there could be text inside the comments that looked like a lambda. To prevent this we used the find function, one for each type of comment used in C++, “/” and “/*”. If the find function finds a match with “/” all characters after it is skipped on that line. If regex matches a “/" all characters after and lines until the end of the comment, “/*" are skipped. Code before each type of comment is still matched to see if there’s a lambda. To prevent code that looks like a lambda but is not a regex was created that looked for the words “operator”, “delete” and “new” followed by the initial part of a lambda, if the regex finds a match, this line is skipped. Characters in strings could also look like a lambda, to prevent this the find function was used. Find searched for the first and the last “" on the current line, and if they existed, it removed all the characters between the quotation marks. The lambda detecting regex was looking for all possible characters that could be found before the initial part of a lambda. Then the initial part and all characters inside the capture. Last was the start of the params part of a lambda. If a lambda was found, the tool would print the path to the current file and the current line to a file. When all repositories are analysed, the name of each repository containing a lambda is printed to a separate file and are extracted from the initial list of all repositories. This leaves one file with all repositories utilising lambda expressions and one file with all

![Fig. 7.2. Visualisation of lambda detector functionality.](image-url)
repositories not utilising lambda expressions. The functionality is summarised in Fig 7.2.

2.2. Comment Counter

Comment Counter uses the output from Lambda Detector to approximate if the lambda has a comment above, inside the lambda or if it is a one-liner, after the lambda. This method for checking if the lambda is commented is adapted from [8] where they examined if a lambda in Java is commented above or inside the lambda. We implemented a C++ variation of their method to make comparisons between C++ and Java. The tool uses the path and line number from the Lambda Detector output. It then begins to read from one row above the match to see if there’s a comment above the lambda. From hereon a distinction is made between one-liners and multi-liners. The difference is that one-liners are lambdas that have their ending and starting braces on the same row, whereas multi-liners have the ending brace on another row. To decide when a multi-line lambda ends the occurrence of curly braces is counted. Each starting curly brace adds one to the total count while each ending curly brace subtracts one from the counter. The curly braces are considered to have been matched when the counter is zero or less. In Fig 7.3, it is shown where comments in the vicinity of lambdas are checked after with the tool.

```
One-liner
// above
[](){ /*inside*/}; // after

Multi-line
// above
[](){
... /* inside */
...
};
```

Fig. 7.3. Where the comments are checked after with Comment Counter for these two types of lambdas.

2.3. Developer Experience Measurer

The construction adapts the method used in [30] to measure experience through a commit’s age. It measures from the initial commit by the developer to the last commit by the same developer. To measure the developer experience, we constructed a tool utilising the LibGit2Sharp library. LibGit2Sharp emulates the behaviour of git commands that are run on the command line. The library allows you to examine repositories and their commits programmatically, which has the benefit of being faster than running a shell script with git commands. The library also made it possible to run these types of commands on multiple threads to concurrently query multiple repositories. The program took the output from the lambda detector and queried each listed repository from the set of repositories we had cloned (see Fig 7.4). First, each unique author’s name was copied from all commits for the main branch of the repository into a unique set. Then, the set with author names was iterated, looking for the first and last

---

1 https://github.com/libgit2/libgit2sharp
commit by each author. The time was retrieved in the form of Unix timestamps. The timestamp of the author’s first commit was subtracted from the timestamp of the author’s last commit and the result was examined. If it was zero, then it had to have been the same commit that was found with both searches and thus, it did not meet our criteria. If the resulting value was less than 86400 (the length of a day in a Unix timestamp) then a day had not passed between the commits, thus the experience in days would have been zero which we chose to explicitly not include in our count of the number of considered committers. After that, the value in days was added to a total and the number of considered authors incremented with one. Once all authors had been examined for their experience values, then the total number of days was divided with the number of considered authors, giving a mean experience value in days for the specific repository. The name of the repository and its mean experience value was written to a file.

3. **Categorise, Analyse & Measure**

This section details how the categorisation was made, how all lambdas were analysed and how the developer experience and unit test values were measured and compared.

3.1. **Categorise**

When we gained access to the lambdas present in the repositories, we went through the output and observed how the lambdas were used. The categories were formulated and justified during this process and named after their usage by how they are introduced and used within the code. The aim was to create as many coherent categories as needed to cover the distinct usages of lambdas. From this observation we formulated the following categories:

1. Function Invocation
2. Assignment
3. Used as a function
4. return statement

For each of these groups we formulated rules for when a lambda would be included in each, single category. Function invocation required the lambda to be created as an argument to a function or constructor. The assignment required that the created lambda was assigned to a variable. Used as a function required the lambda to be created and ran in-place. Return statement required that the lambda itself was returned from a function. Table 7.2 shows simplified examples of each category group:

<table>
<thead>
<tr>
<th>Line</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>myFunction(<a href=""> </a>{});</td>
</tr>
<tr>
<td></td>
<td>Created and used as an argument to a function.</td>
</tr>
<tr>
<td>2</td>
<td>auto myLambda = <a href=""> </a>{};</td>
</tr>
<tr>
<td></td>
<td>Assigned to a variable.</td>
</tr>
<tr>
<td>3</td>
<td><a href=""> </a>{ }();</td>
</tr>
<tr>
<td></td>
<td>Ran in-place.</td>
</tr>
<tr>
<td>4</td>
<td>return <a href=""> </a>{};</td>
</tr>
<tr>
<td></td>
<td>Returned from a function.</td>
</tr>
</tbody>
</table>

Table 7.2. Simplified examples of lambda usage of our chosen categories.

For our categorisation we used the top 10 repositories by star count from the RepoReaper dataset. From these ten repositories all 2048 lambda matches were analysed and placed in their own definitive category. This categorisation was used to answer how lambdas were used in the C++ repositories that we analysed.

3.2. **Analyse**

The output of Lambda Detector was manually analysed by reading the source code for the lambdas present in the top 10 repositories. Each match was given a number matching a category present in Table 7.3. To keep track of which match had received which category we used a code editor with line numbering. Each lambda was compared to the categorisation rules and then placed in their intended group by entering a number into the categorisation file. The result was then obtained by counting the occurrence of each number in the categorisation file.

3.3. **Measure**

**Developer Experience**

We measured the developer experience with the Developer Experience Measurer tool. It queried the repositories for their mean developer experience values and output the values into a file. The measurement was done twice, one for the repositories using lambdas and one for those that did not. The mean values of the repositories were then compared by their utilisation of lambdas in a bar graph. The percentage in the bar graph was calculated separately for both groups with (Amount of repositories (Lambda or NoLambda)) / (Total amount of repositories (Lambda or NoLambda)) * 100. The values in the bar graph were placed into intervals of 500 days of experience, up to 2000 days. Outliers with mean values above 2000 days were placed into their own group.
Unit Test Value

To get the unit test value regarding every repository, the cloned repositories names are matched against the repository name in the RepoReaper dataset. The result is then categorised in lambda and no lambda repositories. If there are duplicates, the owner in the database will be matched with the owner of the cloned repository. Each repository is then placed in an interval between 0 and 1 with .1 difference, 0.1 - 0.2, 0.2 - 0.3 … 0.8 - 0.9, 0.9 - 1.0.

4. Validity Threats

The results of this study rely on the methods used which, if they are not accurate enough, may negatively affect the outcome and conclusions of the study.

4.1. Internal validity

Developer Experience Measurement

Measuring developer experience by a developer’s first and last commit may not accurately portray the developer’s actual experience as a developer, but may serve as an indicator of how committed they have been to a single project. We assumed this would translate into the experience within a specific repository. Thus we assumed that the greater the mean experience value for a whole repository, the more longtime committers the repository has. However, this has the downside that if there are many contributors that push commits one day apart and never again then the value will be adversely affected. However, since these contributions have made it to the master branch of the repository of an engineered software project, then the contributor may have made a change that may have been accepted by a peer.

Comment location

When obtaining all comments, there is no way of actually knowing if the comment is about the lambda, the function the lambda is used in or if there’s any relation to the lambda at all. To prevent this all comments could have been manually checked. However, it is not simple to understand the developer’s intention with a comment, which would have ultimately made the results subjective and built on guesswork.

4.2. External validity

Size of sample sets

The size of our sample set with 500 repositories may not be enough to accurately portray how frequently lambdas are used overall, but still give a sizable depiction of the situation. To mitigate the small number of repositories we valued accuracy within the set we had, therefore we made the results within the set as correct as possible by iterating on the tools we constructed. We also selected the repositories by the inherent ordering of the dataset to prevent selection bias. Further, using only 10 repositories for the categorisation of lambdas may give a skewed view of the open source C++ community as a whole. However, we mitigated this by choosing the repositories by their star count, since they are indicative of popularity.

Age of the dataset used

Using a dataset that is already a few years old for selecting our repositories runs the risk of working with outdated data. The state and values would probably be different if a new dataset was created today. However, using an old dataset did not mean that the repositories we were working with were not up to date. The main point in using the dataset was to mitigate repositories that did not contain software engineered projects, also known as noise. However, not all repositories from the RepoReaper dataset could be cloned since they no longer existed, another repository with the same name had already been cloned or the cloning resulted in other errors.

The measurement is also dependent on time. Unless you have the exact same repositories from the exact same time they were cloned, you will most likely receive different but similar results using our method. This depends on the commit activity within the repositories, which may or may not be actively contributed to. To prevent this we ran all tools on one computer where we had the cloned repositories so that we would receive a single, unified result.

VIII. RESULTS

Our approach was to analyse 500 C++ repositories containing software engineered projects. We constructed a set of tools to find lambdas, to find comments above or inside the lambda body and measure developer experience in a repository. In this section we will detail the outcome and results from using these tools and the subsequent manual analysis.

To help answer how lambdas were used in C++ for RQ1 we analysed the selected repositories with the Lambda Detector tool. Out of the repositories analysed, 175 contained lambdas and the remaining 325 did not contain a lambda. A total of 24764 lambdas were found. In Fig 8.1 the percentage of lambda usage is shown. A third of the repositories contain at least one lambda expression, while the majority of the C++ repositories do not utilise lambda expressions. The most common usage for lambdas was in function invocations, with assignment coming second. This is further shown in Table 8.1, which illustrates how 2048 lambdas in the top 10 repositories were categorised.
Lambda Detector Tool used on 500 C++ repositories where 175 repositories contained at least one lambda and 325 repositories did not contain a lambda.

<table>
<thead>
<tr>
<th>Category</th>
<th>Amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Function Invocation</td>
<td>1867 (~91.16%)</td>
</tr>
<tr>
<td>2. Assignment</td>
<td>165 (~8.05%)</td>
</tr>
<tr>
<td>3. Used as a function</td>
<td>8 (~0.39%)</td>
</tr>
<tr>
<td>4. return statement</td>
<td>9 (~0.44%)</td>
</tr>
<tr>
<td>Total</td>
<td>2048 (100%)</td>
</tr>
</tbody>
</table>

Table 8.1. The result from manually analysing and categorising the 2048 matched lambdas from the top 10 repositories for their lambda usage.

The Comment Counter Tool was used to answer how lambda usage may affect comment presence in RQ2. The tool counted the occurrence of comments out of the total amount of lambda matches. Roughly 19% of the lambdas had a comment one row above, inside the lambda body or after the lambda if it was a one-liner. The percentages of lambdas containing comments are shown in Fig 8.2.

A comparison between the amount of comments in C++ and Java can be seen in Fig 8.3. The data for Java is from Alqaimi, Thongtanunam and Treude’s study [8] that measured how lambdas were commented, either above the lambda, inside the lambda body or both. This study applied the same way of measuring on C++ repositories to be able to compare. The bars in the graph show the percentage of lambdas that were commented in a specific way out of the total number of found lambdas for each study. Table 8.2 shows the values that were used in the comparison and to calculate the percentages for the graph. A total number of ~18% of the lambda expressions were accompanied with a comment, when not counting comments after one-liners. Comments on the same row after a one-liner amounted to 524 (~1%).

Table 8.2 shows the values that were used in the comparison and to calculate the percentages for the graph. A total number of ~18% of the lambda expressions were accompanied with a comment, when not counting comments after one-liners. Comments on the same row after a one-liner amounted to 524 (~1%).
The Developer Experience Measure Tool was used to answer how developer experience may correlate with lambda usage in RQ3. The values were then placed within intervals of 500 days. Fig 8.4 shows the distribution of repositories by percentage within each group. Out of the set, repositories with lambdas are more distributed in the higher intervals than their counterparts.

![Developer Experience Value in C++ Repos](image)

**Table 8.2:** The values for commented lambdas from Alqaimi, Thongtanunam and Treude’s study [8] alongside the findings in this study, when measured the same way in C++ repositories.

<table>
<thead>
<tr>
<th>Location</th>
<th>Count</th>
<th>% of total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Language</td>
<td>Java [8]</td>
<td>C++</td>
</tr>
<tr>
<td>Above</td>
<td>1531</td>
<td>1673</td>
</tr>
<tr>
<td>Inside</td>
<td>1298</td>
<td>2286</td>
</tr>
<tr>
<td>Above &amp; inside</td>
<td>258</td>
<td>432</td>
</tr>
<tr>
<td>Total commented lambdas</td>
<td>3087</td>
<td>4391</td>
</tr>
<tr>
<td>Total lambdas found</td>
<td>54071</td>
<td>24767</td>
</tr>
</tbody>
</table>

![Unit Test Value in C++ Repos](image)

To help answer how the presence of unit testing is affected by lambdas in RQ4 we compared lambda usage against the recorded unit test values for each repository. Fig 8.5 depicts the number of repositories with and without lambdas in each specific unit test value interval. Fig 8.6 illustrates these values in two groups, those which have a value of zero and those that use unit testing to some degree. Out of all repositories not utilising lambdas there’s roughly the same amount of repositories that have a unit test value above zero. Compared to repositories utilising lambdas there’s almost double the amount that also has a unit test value higher than zero.

![Unit Test Value in C++ Repos](image)

**Fig. 8.5.** The unit test value from 500 repositories, extracted from the RepoReaper dataset.

**Fig. 8.6.** The unit test value from 500 repositories from the RepoReaper dataset. The unit test values are summed into two groups by their utilisation of unit testing.

**IX. Discussion**

In this section we answer each research question while also comparing the result to previous studies. Further, we talk about our constructions and their functionality compared to other studies methods but also how they could be improved even more.

1. **Research Questions**

   **RQ1:** How does the usage of lambda-functional programming in open source software in C++ differ from Java?

   A study about lambda usage in Java depicted that 11% of the repositories they analysed contained a lambda (2017) [8]. Another study measured this to 19.2% for Java and 53.8% for C++ (2019) [9]. This may help the claim that lambdas are used more in C++ than in Java. However, the result of lambda occurrence in repositories is considerably...
higher for C++ in [9] compared to our results. This may be explained by the differences in methodology since Nielebock, Heumüller and Ortmeier cloned the top 1000 C++ repositories by star count instead of sampling 500 from a dataset. Further, they analysed the programs using parse trees instead of scanning the source code for occurrences. Despite this difference, the information depicts that there are differences in feature usage between the programming languages, which may be of concern for a language designer. It should be noted that C++ has included lambdas for longer and supports the auto construct that is used to deduce the types for the lambda automatically, while Java developers need to rely on a functional interface and explicitly mention types when constructing lambdas. However, the usage areas for lambdas are similar between the languages. In both languages the main use for lambdas is to pass it as an argument to another function, possibly for behaviour parameterisation [7]. It implies that the feature’s inclusion as an anonymous function into the languages was a necessity and has been well motivated by their current usage areas. However, our categorisation efforts were limited due to its manual nature to the top 10 repositories out of the set, which may not be indicative of the open source community usage of lambdas as a whole. However, when the lambda expressions were manually analysed no false or erroneous matches made by the Lambda Detector were encountered.

RQ2: **How does the usage of lambda-functional programming affect in-code comments in open source software in C++ compared to Java?**

The majority of lambda expressions in this study were not accompanied by a comment. This result is similar to Alqaimi, Thongtanunam and Treude’s [8] that came to the same conclusion. However, the amount that is accompanied by comment is more than tripled compared to Alqaimi, Thongtanunam and Treude’s results [8]. This may indicate that lambda expressions in C++ are more often accompanied by a comment compared to Java. The method used in this study also includes comments after a one-liner lambda expression and found that 1% of all lambdas were commented this way. Therefore it may be worth analysing the comments after the lambda expression and also include comments after the lambda expression in further studies. The main finding would be that comments close to a lambda expression are relatively uncommon, since the result indicates that a small number of lambdas are accompanied by a comment. However, the amount of lambda expressions accompanied with a comment is more common in C++ compared to Java. It is problematic to decide whether the comment is considered to be about the lambda expression or the invoking function, which is the main usage area for lambdas. This opens up for further research regarding lambdas accompanied by a comment and their purpose and also, how common it is to comment code overall.

RQ3: **How does the developer experience correlate with the choice to include lambda-functional programming in C++?**

A majority of the repositories had a mean experience value under 500 days. However, the results indicate that repositories having a mean experience value of over 500 are more likely to use lambdas. It may give support to our hypothesis that repositories with lesser experienced developers committing may be less likely to use lambdas. It may also depict that lesser experienced developers just don’t know how to use lambdas, which may be related to Uesbeck et. al. results that previous experience is linked against the ability to solve problems with lambdas in C++ [6]. Further measurements on a larger set of repositories would be necessary to see if our results may be generalisable. It may be of interest to measure if repository experience is also linked to lambda usage in Java. However, the usage and necessity of lambdas may depend on other parameters such as the purpose, size and amount of committers in a repository.

RQ4: **How does unit testing differ between C++ repositories which are either utilising or not utilising lambda-functional programming?**

The result indicates that repositories utilising lambda expressions are more likely to use code considered test code compared to repositories not utilising lambdas. Mazinian et. al. [7] imply that 60% of all lambdas in Java are used in test code. Our results may support the notion, since it seems that if you are using lambda expressions the repository is more likely to have unit testing to some degree. There may be a connection between lambda expressions used in test code also for C++, which may open up for further research regarding whether lambda expressions are more suitable or useful in test code. Since lambda functions were considered too simple or reliant on other functions to be tested, it is possible that they are instead used as a tool in testing. This would correlate well with lambda expression’s anonymous nature and use for functions that are not worth naming.

2. **Tools**

Alqaimi, Thongtanunam and Treude [8] found Java lambdas by looking for an arrow “->” in the source code of the repositories. When they manually analysed lines containing arrows they found that 35% of the matches did not contain an actual lambda expression. Instead it was possible for them to be parts of strings or comments. This study had a similar experience. Therefore a method for removing what's inside comments and strings and the skipping of multiline comments was constructed. Another
source of false matches in C++ was arrays, operators and keywords that contained a similar structure to the start of a lambda. A regexes was used to filter these out. However, the accuracy of our method compared to tools that construct Abstract Syntax Trees (AST) out of source code haven’t been compared. At first an attempt was made to leverage the AST capabilities of Clang\(^2\), since it was assumed it would be an accurate way of discovering lambda expressions. However, the unpredictable and varying nature of the source code in the repositories made using the tool difficult. This was due to the amount of fatal errors when attempting to create ASTs for the repositories. Fixing these problems would have taken a considerable amount of time. A conclusion was made that Clang was not a suitable tool for statically analysing multiple repositories with great variance in programming style, but rather a suitable tool for analysis when developing a single project. Therefore, it was necessary to construct a new tool to scan repositories for lambdas. The tool could be improved by constructing a better regex that finds all possible arrays and has a way to differentiate them from lambdas. While the tool works for the chosen 500 repositories in the RepoReaper dataset, it may match arrays created differently in other repositories.

3. Implications

The results from this study may be implicit feedback to language designers of how much lambdas, as a feature, are used. The low rate of lambda utilisation in Java was surprising when compared to C++. Further research is needed into why this difference exists between C++ and Java. Unfortunately, the quantitative values that were measured and compared do not translate into understanding of the causes. However, the differences in design between these two languages can be used to reason about the causes. There is a need for evaluation of how new features are used to be able to improve their design and make them more attractive for developers. Open source repositories seem like a good source of information for this. However, it is debatable whether having your code public is a form of consent to being used as a source in scientific studies. Further, it can be a sensitive topic to research on the participants in these communities. For instance, calculating the experience value for every author may seem like placing some form of worth on each individual. This form of measurement could be used for malicious purposes, such as creating barriers against contributors that have less experience within a repository.

X. Conclusions

In this study we conducted a systematic static code analysis of C++ repositories from GitHub. The findings indicate that lambdas are used more frequently in C++ compared to Java. The usage areas for lambdas are similar, however lambdas in C++ are more often accompanied with a comment. This study also indicates that a greater mean developer experience and unit test value correlate with the usage of lambdas. These findings aim to address a problem regarding the lack of knowledge of lambda usage in C++ and open up further research. Since lambdas are used more in test code in Java, and this study shows that lambdas in repositories with a unit test value are more common, is it possible that lambdas in C++ also are more common in test code.

References


\(^2\) https://clang.llvm.org
**APPENDIX I: TIME PLAN**

### Initial Plan

<table>
<thead>
<tr>
<th>Task/Week</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Writing theoretical basis in document</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Planning, implementing, testing lambda-tool</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Collecting repositories</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Measuring developer experience</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Comparing unit test inclusion values</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Analysing lambda comments</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Analysing lambda usage</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Finalizing report and preparing presentation</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Actual Plan

<table>
<thead>
<tr>
<th>Task/Week</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Writing theoretical basis in document</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Planning, implementing, testing lambda-tool</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Collecting repositories</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Measuring developer experience</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Comparing unit test inclusion values</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Analysing lambda comments</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Analysing lambda usage</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Finalizing report and preparing presentation</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Every step of the way will include writing of the report. These are rough estimates of how long each step is expected to take, but the timeframes may move.

### APPENDIX II: CONTRIBUTIONS

This is how the major responsibilities were divided among us working on this thesis. It was however natural that we co-operated on each part to troubleshoot and fix bugs.

Jonathan: Development of lambda matching tool, regex construction, background, running and getting results, creation of graphs, unit testing values

Heidi: RepoReaper dataset filtering, Developer Experience Measure tool, Comment Counter tool, categorization, related works, background (only developer experience), other illustrations