ARTICLE TYPE

Embracing Modern C+ Features: An Empirical Assessment on the KDE Community

Walter Lucas*[1] | Fausto Carvalho[1] | Rafael Campos Nunes[1] | Rodrigo Bonifácio[1] | João Saraiva[3] | Paola Accioly[2]

[1]Computer Science Department, University of Brasília, Brasília, Brazil

[2]Federal University of Cariri, Juazeiro do Norte, Brazil

[3]Universidade do Minho, Braga, Portugal

Correspondence
*Corresponding author, Email: walter.mendonca@aluno.unb.br

Summary

Similar to software systems, programming languages evolve substantially over time. Indeed, the community has more recently seen the release of new versions of mainstream languages in shorter and shorter time frames. For instance, the C++ working group has begun to release a new version of the language every 3 years, which now has a greater number of modern C++ features and improvements in modern standards (C++11, C++14, C++17, and C++20). Nonetheless, there is little empirical evidence on how developers are transitioning to use modern C++ constructs in legacy systems, and not understanding the trends and reasons for adopting these new modern C++ features might hinder software developers in conducting rejuvenation efforts. In this paper, we conduct an in-depth study to understand the development practices of KDE contributors to evolve their projects towards the use of modern C++ features. Our results show a trend in the widespread adoption of some modern C++ features (*lambda expressions*, *auto-typed variables*, and *range-based for*) in KDE community projects. We also found that developers in the KDE community are making large efforts to modernize their programs using automated tools and we present some modernization scenarios as well as the benefits of adopting modern C++ features of the C++ programming language. Our results might help C++ software developers, in general, to evolve C++ legacy systems and tools builders to implement more effective tools that could help in rejuvenation efforts.

KEYWORDS:
Language Evolution, Software Rejuvenation, C++ Programming Language

1 | INTRODUCTION

The C++ programming language was designed at the beginning of the 1980s with the main purpose of extending the C language with object-oriented constructs. The first C++ standard was released in 1998 (C++98), with contributions from a language committee[1] formed in 1990. The language has adhered to the C++11 standard in 2011 and is currently receiving updates more frequently (an estimated time of three years difference between releases of the new standards). These updates,

---

[1]Committee ISO/IEC JTC1 (Joint Technical Committee 1) / SC22 (Subcommittee 22) / WG21 (Working Group 21)

in turn, made the language more expressive and secure, adopting certain idioms or completely new constructs such as type inference, lambda expressions, resource transfer, smart pointers, and concurrent support within the standard library[1].

These language modifications change the way developers write their programs. Nonetheless, keeping legacy systems up to date with new versions of a language is a common concern related to language evolution[2]. On the one hand, developers wish to use modern C++ features in their programs, to improve some quality concerns (e.g., software readability, maintenance, security, and so on). On the other hand, migrating a code base to support new versions of a language is everything but trivial. Developers have to conciliate dependencies with external libraries and also consider the implications of changing the configuration or versions of compilers to the users of their programs or libraries[3]. Indeed, we have observed in the developer's mailing lists of open source communities some extensive, even heated discussions about whether or not to conduct a maintenance effort on a program or library towards language evolution. For instance, Figure 1 presents two snippets of message threads from the open source community, revealing not only the interest in updating the code to support C++11, but also some challenges that might hinder software developers from modernizing legacy code bases.

kde-devel mailing list (May, 2022)

Hello, KDE team,

I would like to create the independent fork of Krita that would allow to use all modern features of C++, up till that is supported by the latest GCC release, 12.1 at the time of writing.

The current Krita development rules are limited by C++11 that is now the ten years old standard. Even that is limited by the lengthy list of restrictions. The rules also require using deprecated features of the Qt framework like Q_FOREACH. I understand the care of the community not to spoil anything and to preserve the beauty of the existing code. This means radical changes should be done in a form.

OpenJDK JEP 347: Enable C++14 Language Features (July, 2018)
Summary:

- Allow the use of C++14 language features in JDK C++ source code, and give specific guidance about which of those features may be used in HotSpot code.

[…]

Risk and Assumptions:

- There may be other platforms with toolchains that do not yet support the C++14 language standard.

- There may be bugs in the support for some new features by some compilers.

FIGURE 1 Open source forum messages discussing the need for updating the C++ version

Given the importance of C++ and the challenges related to evolving legacy systems to adopt new versions of programming languages, it is worth characterizing how software developers embrace the modern features of C++ and which practices developers use during rejuvenation efforts. Indeed, previous work[4,5,6] reports efforts to *rejuvenate programs*, a particular kind of software maintenance whose goal is to replace legacy code features and idioms with modern constructs available in recent versions of a programming language[7]. Efforts to rejuvenate software systems have also been investigated and reported in the literature.

For instance, Mazinanian et al.[5] report a large-scale study to understand how professional developers use lambda expressions in Java programs, after its adoption on Java 8. The study evaluated 100,540 lambda expressions in 241 open-source projects, revealing an increasing trend in lambda expression adoption. Similarly, Alqaimi et al.[4] investigate the use of lambda

expressions in Java programs on GitHub and report that 11% of the repositories use lambda expressions and that only 6% of lambda expressions are accompanied by source code comments—which might suggest that lambda do not make the programs harder to understand. The authors also present an automated tool to generate complete documentation for lambda expressions in Java. Kumar et al.[6] present a tool for rejuvenating C++ code that implements source code transformations of C++ macros into C++11 feature usages. They report that 68% to 98% of macros can be translated into modern C++11 features. In addition, the authors also present techniques to assist developers to automate the rejuvenation process.

Despite all these research efforts to discuss modern C++ features adoption, there is still a lack of literature about the practices C++ developers use to conduct rejuvenation efforts. Our research aims to fill this gap by conducting a large-scale empirical study where we analyze 272 open source projects from the KDE community [2], in order to understand the trends in the adoption of modern C++ features and the practices developers use while modernizing C++ code. Our findings characterize maintenance efforts in the KDE community to introduce modern C++ features released in the recent versions of C++: C++11, C++14, and C++17. We discuss the trends in the adoption of modern C++ features such as *lambda expressions*, *auto-typed variables*, and *range-based for*, and our results bring evidence of the use of automated tools for C++ code modernization. In addition, we present a catalog of commits 1 of rejuvenation efforts conducted by KDE developers that could help other developers rejuvenate their programs and tool builders identify opportunities to implement source code transformations.

This paper is organized as follows: Section 2.1 presents some background information about the evolution of C++ language. Section 2.2 compare our study with previous work about software rejuvenation in the literature. Section 3 presents our research questions and the procedures that we follow during our investigation. Section 4 shows the results of our quantitative assessments and its implications. Section 5 presents some examples of rejuvenation efforts and summarizes the perspective of KDE developers rejuvenation efforts in C++ project. In Section 6 we summarize the implications of our study and the main threats to the validity of our work. Finally, Section 7 presents our final considerations.

Artifacts availability. Our study is fully reproducible. All developed tools, collected data, and R scripts are available online for statistical analysis in a Git repository: https://github.com/PAMunb/cppEvolution

2 | BACKGROUND AND RELATED WORK

Some programming languages are constantly evolving and follow a rigorous and organized evolution process. The C++ community, for instance, is responsible for steering the language's evolution in accordance with the recommendations of the International C++ Standard Committee. The C++ community is comprised of organizations such as Apple, Facebook, Google, IBM, Intel, Microsoft, and Nvidia, as well as interested parties from multiple nations. In addition, universities and hardware vendors support the C++ committee, along with representatives from diverse fields such as finance, game development, C++ library communities (e.g., Boost), and platform vendors. The community organizes meetings into working groups (WGs) and study groups (SGs)[1]. It is the responsibility of committee members to submit proposals, which may include modifications to the language or standard libraries. Additionally, the committee functions as a filter to prevent bad proposals from entering the standard[1]. The majority of the committee's efforts are devoted to addressing simple issues such as naming conventions, grammatical details, the addition of new constructs, and backwards compatibility with previous versions of C++. In the following Section 2.1, we provide an overview of the C++ evolution process and its modifications. Next, in Section 2.2, we discuss related works and how our own compares to them.

2.1 | C++ Language Evolution

The C++ programming language is a multi-paradigm and general-purpose language, whose first official version, known informally as C++98, has been released as an ISO/IEC standard (ISO /IEC 14882:1998)[8]. The C++98 standard includes numerous language features such as templates, exceptions, dynamic_cast, namespaces, declarations in conditions, named casts, and bool_type. C++ also includes the Standard Template Library (STL)[9], which provides features such as the general and efficient framework of containers, iterators, and algorithms. In addition, other resources such as traits, string, bitset, locales, auto_ptr, and shared_ptr were added to the standard language library[10].

---

[2]KDE is an international cross-platform project community designed for Linux systems.

In 2003, the WG21 committee issued a new version, which became known informally as C++03. This version of the language was primarily a bug fix release that included several library revisions (C++ Standard Library Issues List TC1 [3]). In 2006, the committee issued a new version known as C++0x. The C++0x version was to contain a large number of features that were frozen and not released until the next version in 2011 [1]. Starting in 2011, the International C++ committee agreed to publish a new C++ release every 3 years.

In 2011, the committee released the standardized version known as C++11, which brought significant changes to the C++ [1] programming language and paved the way for what is now called *Modern C++* [11]. C++11 introduced various modern language features such as auto and decltype, range-for, nullptr, constexpr functions, user-defined literals, lambda expressions (unnamed function objects), variadic templates, and the noexcept keyword, among many others [12]. The C++11 version also included several built-in modern C++ features from the Boost Library [13], which had a significant impact on the new release of the C++ [14] Standard. Some of these modern C++ features are the thread library, exception_ptr, error_code, error_condition, and iterator improvements.

The C++14 standard was issued by the WG21 committee as an extension of the C++11 [1]. The changes consisted mainly of bug fixes that had been noticed during the initial use of the C++11 standard, and also included minor improvements to existing language features (two-range overloads for some algorithms, type alias versions of type traits, user-defined literals for basic_string, duration, and complex) [12]. The C++14 version included language features such as variable templates, generic lambdas, lambda init capture, new/delete elision, and aggregate classes with default non-static member initializers [12]. Some library resources such as std::make_unique, std::shared_timed_mutex, std::shared_lock, std::integer_sequence, std::exchange, and std::quoted have also been included.

The C++17 standard has not undergone major changes like the previous standards C++11 and C++14, but it includes modifications and features to improve the previous standards. C++17 included changes to remove language and library features that were deprecated during the C++ standard evolution [11]. C++17 also added modern language features like u8 character literal, made noexcept part of the type system, new order of evaluation rules, and lambda capture of `*this`. Improvements in the library were also part of this standard, such as file system, scoped_lock, shared_mutex (reader-writer locks), any, variant, optional, string_view, parallel algorithms, and others [11,15].

C++20 introduced a programming concept that differs from previous standards, with language features aimed at concurrency, generic lambda expressions, metaprogramming, stricter type safety, and others [16]. The new standard has language features such as feature test macros, the 3-way comparison operator <=> and the operator == () = default, designated initializers, init statements, and initializers in range-for. C++20 also included new library features such as concepts, ranges, dates and time zones, span, formats, improved concurrency, parallelism support, and others [1].

2.2 | Research on Software Rejuvenation

Software maintenance can be motivated by various factors, such as business adaptations, changing requirements, architectural transitions (e.g., the use of cloud environments), the need for quality improvements (e.g., refactoring), and also technical aspects, such as program evolution scenarios that are driven by the evolution of the programming languages used [17]. Software rejuvenation is a particular type of software maintenance whose goal is to replace obsolete features with modern programming language features that can coexist with evolved software [2]. Research works that investigate the evolution of software through rejuvenation is increasingly common in software engineering [6,5,18,19].

Regarding previous studies proposing tools to automate software rejuvenation, Kumar et al. [6] presented a tool for rejuvenating C++11 code. The methodology proposed in the paper involves three main steps: identifying preprocessor macros in the codebase using a combination of lexical and syntactic analysis, transforming the macros into equivalent C++ code using a set of predefined rules, and analyzing the transformed code to detect any issues that the macro transformation process may have introduced. They reported that modern C++11 features can replace 68% to 98% of macros. Their results suggest that it is possible to replace a large part of legacy code with modern C++ features, but some transformations require manual adjustments. Dantas et al. [18] presented a library of Java transformations developed in the Rascal meta-programming language to rejuvenate legacy systems to support newer programming language constructs. The authors ran a total of 2,462 source code transformations in 40 open-source projects. A small sample of these transformations was submitted to projects via GitHub's pull-requests mechanism. The study identified that simple transformations, such as adopting *diamond operators*, are more

---

[3]https://open-std.org/JTC1/SC22/WG21/docs/lwg-status.html#TC1

likely to be accepted than transformations that substantially change the code, such as replacing *for loops* by the new functional style. Unlike our study, these works[6,18] do not investigate what motivations drive developers to adopt modern language features or the benefits of adopting modern language features.

Mazinanian et al.[5] conducted a large-scale study with a mixed-methods approach to understanding how developers use lambda expressions in Java programs. The authors evaluated 100,540 lambda expressions across 241 open-source projects to identify the usage patterns of lambda expressions. They also surveyed Java developers to collect information on their experience with using lambda expressions. The survey questions focused on developers' experiences with the language feature, how often they use it, and what benefits and limitations they perceive when using it. The study revealed an increasing trend in adopting lambda expressions in 2016 (the proportion of lambdas introduced per line of code doubled compared to 2015). In addition, they present that leading developers introduce more lambda expressions than external collaborators and that Java developers tend to write them manually. We also found a trend (Section 4) of adoption of some features (*auto-typed variables*, *lambda expressions*, and *range-based for*) since 2016 for C++ projects in the KDE community. In addition, our study shows evidence that the top developers were responsible for 63.2% of the rejuvenation commits found. Finally, the survey results indicate some benefits of using lambda expressions, like improved code readability and reduced code verbosity. However, developers also reported challenges when using lambda expressions, including debugging issues. Additionally, the findings suggest that the Java community should continue to improve tooling to support developers' use of lambda expressions.

Lucas et al.[19] conducted an empirical study to investigate the impact on code comprehension after introducing *lambda expressions* in Java programs. The study evaluated 66 pairs of code transformations (one pair consists of a source code snippet in the version before the transformation and the same snippet after the introduction of *lambda expressions*). First, some metrics were extracted for each pair of transformations: two metrics related to source code complexity (number of lines of code and cyclomatic complexity[20]) and two metrics that estimate source code readability[21,22]. Then, the authors surveyed professionals to collect their perceptions about the benefits of understanding the program by adopting *lambda expressions*. Such results presented a contradiction. Based on the quantitative assessment, the researchers found no evidence that the introduction of *lambda expressions* leads to improvements in the readability of software, which differs from the qualitative evaluation that suggests improvements in program comprehension. Additionally, the study revealed that the use of *lambda expressions* in some scenarios can improve code comprehension (like the replacement of anonymous inner classes by *lambda expressions*). Differently, just replacing a simple `for statement` over a collection statement by a `collections.forEach()` does not bring any benefits, according to the participants of the survey. Our work differs from those studies[5,19] in two aspects: (a) we study the adoption of modern features introduced by the evolution of the C++ language introduced between three versions (C++11, C++14, and C++17); and (b) our study is not limited only to features that introduce the functional style (such as lambda expressions).

Contrary to Mazananian et al. and Lucas et al., Uesbeck et al.[23] and Zheng et al.[24] conducted studies showing harmful effects of adopting *lambda expressions* in C++ and Java projects, respectively. Uesbeck et al. conducted controlled experiments to understand the impact on developers' productivity while implementing different tasks with and without *lambda expressions*. Their results show that using *lambda expressions* negatively affects productivity for less experienced developers regarding how quickly they can write correct programs. However, it had no positive or negative effect on more experienced developers' productivity. The results suggest that learning how to use *lambda expressions* properly might take some time. Moreover, Zheng et al. show data confirming a trend of developers removing *lambda expressions* from their code in Java starting in 2016. They conduct a study to understand the reasons behind such phenomena. As a result, the authors describe developers often misusing *lambda expressions*, causing unwanted side effects, such as memory leakage or inducing new bugs. They provide seven main reasons for removing *lambda expressions* and seven common migration patterns. Such results are beneficial for developers that are still learning how to use lambda expressions in Java. These previous studies[23,24] complement our results since we also show an increasing use trend in lambda expressions since 2016 and the factors that might explain it, but we do not analyze the harmful effects of this trend.

Finally, Chen et al.[25] conduct a study to understand how developers have used C++ templates since their first release. They find out that the main reason for using C++ templates is to avoid code duplication. Moreover, they also measure the proportion of explicit type declarations versus implicit type declarations (the *auto-typed variables*), concluding that developers prefer to keep explicit type declarations instead of changing to the auto type variables. This result complements our study because, while we describe many examples where developers chose to rejuvenate code by using *auto* type variables, we do not measure how much of the code still uses explicit type declarations.

3 | STUDY SETTINGS

This study aims to gain a comprehensive knowledge of how C++ developers utilize *modern* C++ features included in the C++11, C++14, and C++17 standards. This study focuses on a few of C++11 and C++14 innovations, including *lambda expressions*, type inference algorithms based on *auto-typed variables*, and the brand-new *range-based for* statement. According to Stroustrup, these features would be subject to more adoption expectations[1]. In addition, we incorporate the C++17 *if-statement with initializer* and the new C++ support for concurrent programming in our research.

Also related to the scope of our research, we focus on a relevant organization of C++ developers: the KDE open community, an international free software community responsible for developing hundreds of applications targeting different domains, such as Games, Education, and Frameworks; including the Plasma desktop that runs in Linux distributions (e.g., OpenSuse) and mobile phones. Similar to other open source organizations, KDE makes available development policies[4] and leverages static analysis procedures for conformance checking (including the Krazy tool[5]).

We started analyzing 1061 C++ repositories that we checked out from the GitHub KDE community (the official read-only mirror of the KDE projects). We removed eleven repositories that do not contain C++ files. From the remaining 1050 repositories, we filtered out projects with a small percentage of C++ code (below 50%) and projects that started after 2010 or that did not have recent updates—that is, we only consider projects that have at least one commit in 2022. Our curated dataset contains 272 KDE programs and libraries written in C++.

3.1 | Research Questions

We investigate the following research questions:

(RQ1) To what extent do KDE systems rely on modern C++ features?

(RQ2) When did KDE developers start using modern C++ features?

(RQ3) Is there any trend in the adoption of modern C++ features in KDE applications?

(RQ4) Do KDE developers conduct maintenance efforts having the sole goal of rejuvenating C++ code?

(RQ5) Which tools do KDE developers use to support maintenance efforts for code rejuvenation?

(RQ6) What are the reasons that motivate KDE developers to conduct maintenance efforts for code rejuvenation?

(RQ7) Are the Core Developers of the projects responsible for conducting rejuvenation efforts in KDE projects?

Answering the first research question allows us to understand how popular modern C++ features such as *lambda expressions*, *auto-typed variables*, *range-based for*, and *if-with-initializer statements* are in the KDE applications' code base. Answering the second research question enables us to know how long it takes for fully fledged KDE projects to start migration efforts of the codebase to use modern C++ features. Answering the third research question allows us to understand if there is a trend toward rejuvenating the code of KDE applications. Answering the fourth and fifth research questions allows us to identify whether or not KDE developers conduct rejuvenation efforts, and, if so, what tools they use. Answering the sixth research question allow us to identify what motivations lead KDE developers to conduct rejuvenation efforts in their programs. Finally, answering the last research question enables us to identify which developers conduct rejuvenation efforts and whether they are the main developers of KDE projects.

3.2 | Research Procedures

To answer our research questions, we mined the source code repository of 272 KDE applications and collected facts about the usage of modern C++ features using a static analysis tool we implemented using Java and the Eclipse CDT infrastructure for parsing and traversing C++ code. For the research questions RQ5 and RQ6, we also surveyed KDE developers via e-mail messages.

---

[4]https://community.kde.org/Policies
[5]https://github.com/Krazy-collection/krazy

In more detail, to answer the first research question, we first run our static analysis infrastructure on the latest revision of the applications. We then (i) calculate the absolute number of occurrences of the modern C++ features we are interested in (e.g., *lambda expressions*, *auto-typed variables*, *range-based for*, and *if-with-initializer statements*) and (ii) carry out descriptive statistics on that data. To answer the second and third research questions we collect the same statistics considering a weekly-based interval of all revisions of the applications' code base, starting from January 1, 2010, until June 1, 2022. Our primary motivation for establishing this range is due to the considerable number of commits that some projects have, which can lead to a time-consuming analysis. As a means to mitigate this issue, we defined the range to ensure the feasibility of our study. Algorithm 1 clarifies this approach. We use Time Series to investigate the trend in the adoption of modern C++ features.

Algorithm 1 Reverse walking of the code history (RWCH).

Require: A valid source code Git repository (*repository*)
Ensure: a set (*observations*) whose entries contain the metrics for a given revision in the source code
    previous_date = null
    observations = {}
    for Revision r ∈ repository.history() **do**
        current_date = r.date
        if previous_date == null or diffInDays(current_date, previous_date) >= 7 **then**
            repository.checkout(r)
            metrics = traverse(repository)
            observations = observations ∪ (r, metrics)
            previous_date = current_date
        end if
    end for

We use a conservative heuristic to search for commits that characterize rejuvenation efforts in KDE projects and then answer the fourth and fifth research questions. Our heuristic iterates over consecutive commits in our dataset (i.e., with a distance of at least seven days) and computes the increase in the adoption of each modern C++ feature. Whenever we find an increase of at least 50% in the adoption of modern C++ features and the amount of statements increases by less than 5%, we assume that a rejuvenation effort might have taken place in that period.

Using this conservative heuristic, we automatically identified 207 intervals that could contain a rejuvenation effort. We then retrieve all commits within the period of each interval and again perform a search for patches that might contain a rejuvenation commit. Using our approach, we then iterate over the commits several times, reducing the number of commits that must be manually evaluated. Within all intervals, we found the 207 that contain between 1 and 24 thousand commits. Since this approach would demand a huge amount of time to iterate over large commits to collect the metrics, we use a keyword search in the commit messages in cases where an interval contains more than 100 commits. The keywords we consider in the search include `lambda`, `auto`, `range-based`, or `modern for` combined with `modern`, `modernize`, `port away`, `migrate`, `migration`, `c++11`, `c++14`, `c++17`. We select these keywords from commits we found during a manual search of commit messages. Otherwise, if the interval contains fewer than 100 commits, we use the iterative approach to identify smaller intervals that might contain a rejuvenation effort. Our heuristic produces two independent text files, containing the results of the commit message keyword-based search and the results of the iterative search. We manually analyze the results to validate whether or not a given commit corresponds to a rejuvenation effort.

Subsequently, we mine the authors of the commits that we validate as *rejuvenation efforts*. To answer the sixth research question, we contacted these developers via email to understand their motivations for carrying out maintenance efforts aimed at rejuvenating the projects' source code. The developers we contacted are the same contributors who conducted the large rejuvenation efforts identified in earlier stages of our research. Finally, to answer the last research question, we used the notion of Truck Factor (TF) to verify if these contributors are the core developers of the projects in which they conducted rejuvenation efforts. The notion of TF aims to identify a minimum set of developers that, if they abandon the project, the maintenance and evolution of that project might be discontinued. Furthermore, TF is suitable for assessing the distribution of knowledge among the developers of a project as shown by Ricca et al [26]. As reported by Bosu and Sultana [27], the notion of TF

can also be used to identify developers who have made significant contributions to guide the development and evolution of projects. There are several approaches to calculating Truck Factor in the literature, but here we used the approach proposed by Avelino et al[28] for the following reasons: **i)** the approach proved to be superior to other strategies[29] and **ii)** there is an easy-to-use implementation available, which **iii)** has been used in another study to identify core developers[30].

In Section 4 we present the results of a descriptive statistic analysis of our dataset, considering all projects. In this way, we give high-level answers to our research questions. Next, in Section 5.2, we present more detailed qualitative information regarding the KDE developer's adoption of modern C++ features.

4 | RESULTS OF THE QUANTITATIVE ASSESSMENT

In this section, we present the findings of a descriptive analysis that we conducted on our dataset, which includes all projects. We also address the research questions RQ1, RQ2, and RQ3. In the following section, we will discuss qualitatively the modern C++ features used by KDE developers. Table 1 shows the general adoption of our modern C++ features of interest in KDE projects, as well as when a feature first appeared in our dataset. Notably, *auto-typed variables*, *lambda expressions*, and *range-based for* are present in over sixty percent of the KDE projects in our dataset. This confirms Bjarne Stroustrup's prediction that certain modern C++ features would be extensively adopted[1]. Moreover, a few months before the formal release of C++11, *auto-typed variables*, *lambda expressions*, and *decltype specifier* began to appear. Although C++11 added C++ multithreading capability, it is hardly used in our dataset. Conversely, we did not find evidence of the use for four other modern C++ features (*async function*, *future declarations*, *promise declarations*, and *shared future declarations*) that were in our initial subset. Therefore, we have focused our research on these three modern C++ features (**auto-typed variables, lambda expressions, and range-based for**).

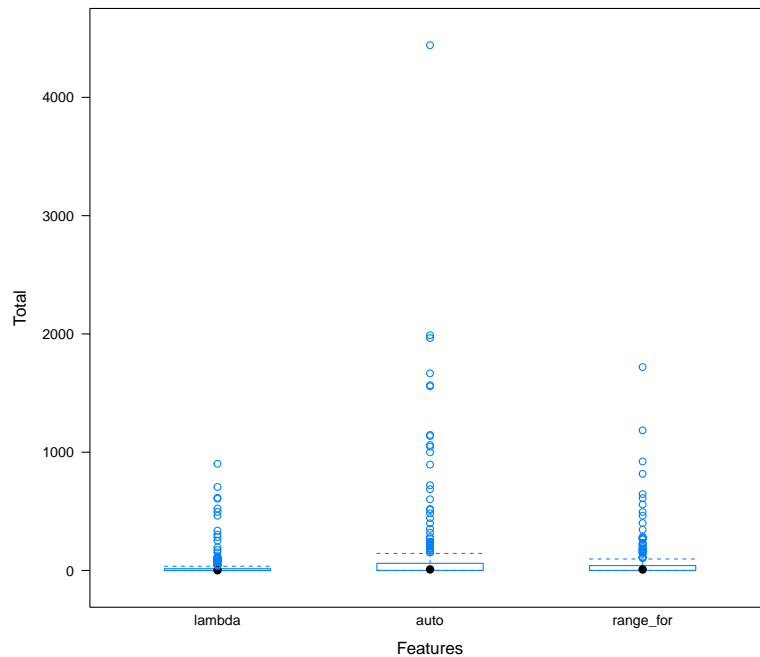**TABLE 1** Adoption of modern C++ features in KDE projects.

Feature	Projects Adoption (%)	Occurrences (#)	First Occurrence
auto-typed variables	80.51	33225	Mar-2010
constant expressions	14.70	777	Nov-2012
if-with-initializer statements	4.4	71	May-2020
lambda expressions	63.60	8918	Mar-2010
range-based for	78.30	16485	Dec-2011
thread declaration	0.73	6	May-2018
decltype specifier	2.2	27	Apr-2010

The modern C++ features *lambda expressions*, *auto-typed variables*, *range-based for* are widely used across the projects, even though their adoption distribution is not uniform (see Figure 2). The median distribution of *lambda expressions*, *auto-typed variables*, and *range-based for* is 2, 9, and 8.5, respectively. However, the KDevelop project alone contains 4441 occurrences of *auto-typed variables* (13.36% of the total). We observe the same lack of uniformity for *lambda expressions* and *range-based for*. We run a test to estimate the Spearman correlation[31,32] between the modern features adoption and the size of the projects (in terms of the total number of statements and the total number of C++ files). First, we found just a moderate correlation ($cor \leq 0.62$) between the adoption of language features and the size of the projects, indicating that we cannot explain a large use of language features using only project size metrics. Next, we run a test to estimate a correlation between the usage of modern C++ features (*lambda expressions*, *auto-typed variables*, *range-based for*). We found a positive and strong correlation ($cor \geq 0.71$) between *lambda expressions*, *auto-typed variables*, and *range-based for* usage and a very strong correlation ($cor \geq 0.85$) between *auto-typed variables* and *range-based for*. Although LLVM introduced the support for modern C++ features before 2011 (e.g., LLVM version of 2008 already supports *lambda expressions*[33]), our results give evidence that, within the KDE projects in our dataset, the widespread adoption of modern C++ feature took place years after the release of the C++11 standard[34].

RQ1: **To what extent do KDE systems rely on modern C++ features?**

Our findings suggest that KDE developers extensively use *auto-typed variables*, *lambda expressions*, and *range-based for*—more than 80% of the projects use *auto-typed variables* in our dataset; while more than 60% of the projects use *lambda expressions* and 78% of the projects use *range-based for*. Other modern C++ features, such as *constant expressions*, *decltype specifier*, *if-with-initializer statements*, and *thread declaration* are rarely used in KDE projects.


FIGURE 2 Distribution of modern C++ features usage across the KDE projects in our dataset. Each of the points on this graph represents the respective amount of that modern feature (*auto-typed variables*, *lambda expressions*, and *range-based for*) usage per project.

Although KDE developers started using *auto-typed variables*, *lambda expressions*, and *range-based for* before 2012, we observed a more significant growth in their usage after 2016 (five years after the release of the C++11 specification.) see Figure 3. After this moment, we found rejuvenation efforts that changed many files to introduce these language features. We present more detailed information about these efforts in the next section.

RQ2: **When did KDE developers start using modern C++ features?**

Our findings suggest that the widespread adoption of modern features happened five years after the release of the C++11 specification—for those modern features frequently used. Accordingly, our results suggest that the general and widespread adoption of new language is not immediate—even for long waited features such as C++ *lambda expressions*.

Besides an observable trend in adopting new C++ features, Figure 3 presents some interesting situations, in which the total number of a given language feature usage declines. We manually investigated this issue and found two main reasons for that: (a) there are specific commits that revert contributions that aim to rejuvenate the code base and (b) commits that merge branches often lead to this disruptive language feature adoption pattern, which contains contributions that reduce the number of feature adoption followed by contributions that rejuvenate the code again.
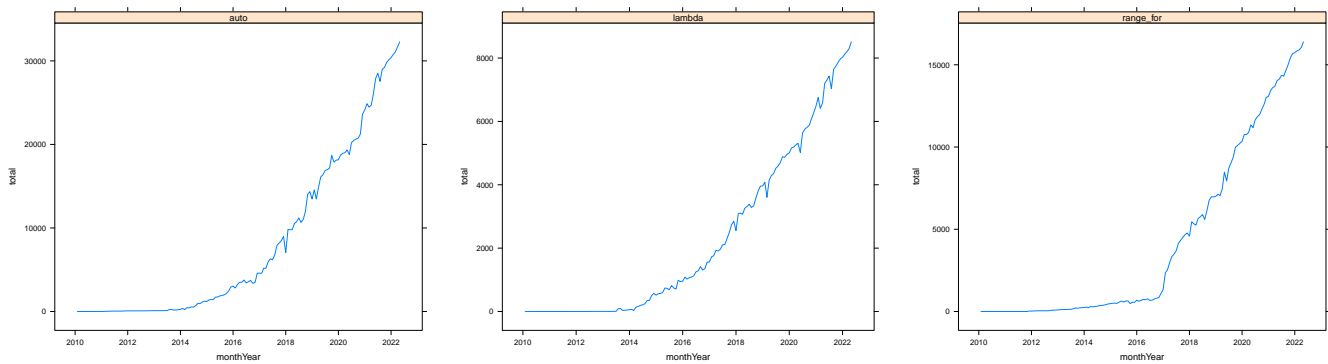
FIGURE 3 Distribution of *auto-typed variables*, *lambda expressions*, and *range-based for* usage across the KDE projects in our dataset.

For instance, commit 1a5e7ab4 of `konversation` (commit date on 1 Oct 2020 and message *Port away from Qt's* `foreach`: *loops over method-local containers*) converts 110 *foreach* statements into the modern *range-based for* statement. This commit changed a total of 41 files. [6] Subsequently, commit ef184b7d reverts the changes in the aforementioned commit. The reversion goal is explicit in the commit message: *"Revert Port away from Qt's* `foreach`: *loops over method-local containers"*. The author of this commit further details his/her motivation: *"This patch reverts commit 1a5e7ab, which was accidentally ... pushed as squashed commit of multiple commits which should be separate."* For this particular case, the motivation to revert the commit was a procedure error when integrating and pushing a set of commits. We also observed the exclusion of many modern feature usages after merge commits.

We model our dataset using *time series* and conduct a regression analysis study using the `tslm` function available in the `Forecast R package`. To this end, we consolidate a monthly snapshot of the total usage of the modern C++ features *auto-typed variables*, *range-based for*, and *lambda expressions*. The `tslm` function implements a conventional regression analysis, though it includes additional predictors such as *trending* and *seasoning*. Since our dataset does not present seasonal data, we just estimate the *trend* component. The regression analysis results suggest a statistically significant trending increase for the adoption of the three features in our study (*auto-typed variables*, *range-based for*, and *lambda expressions*), with trending values of 210.78, 112.48, and 57.19 for *auto-typed variables*, *range-based for*, and *lambda expressions*, respectively. This means that every month, there is a trend of including 210.78 new *auto-typed variables* declarations. Note that if we constraint the same analysis for the period from 2010 until 2016, the same analysis will lead to trend values of 40.02, 10.29, and 14.31 *auto-typed variables*, *range-based for*, and *lambda expressions*. This result suggests a reasonable increase in adopting these modern C++ language features.

> **RQ3: Is there any trend in the adoption of modern C++ features in KDE applications?**
>
> Our findings also suggest a consistent trend towards increasing adoption of modern C++ language features in KDE projects, as well as a leaning toward the migration of legacy code to adopt *auto-typed variables*, *lambda expressions*, and *range-based for*. Nonetheless, occasionally, it was possible to identify commits that regress some migration efforts. This might explain points in the change history of the systems where we observed a reduction in adopting new language features followed by new commits that revert the reduction.

We find this trend interesting, and we investigated the motivations that can lead to increased adoption of these modern C++ features (*auto-typed variables*, *lambda expressions*, and *range-based for*). We present quotes from the answers of KDE developers who participated in our survey and some results of our descriptive analysis that can explain this trend. We detailed it in Section 5.3. In the next section, we present some examples of rejuvenation efforts to introduce *auto-typed variables*,

---

[6]Qt is an application development framework widely used by KDE projects. The KDE Free Qt Foundation https://kde.org/community/whatiskde/kdefreeqtfoundation/ ensures the availability of the Qt tools for KDE software development.

lambda expressions, and *range-based for* statements. In particular, we highlight the KDE developers' perspectives on this particular type of software maintenance.

5 | RESULTS OF THE QUALITATIVE ASSESSMENT

We use two strategies in our qualitative assessment. First, we mine the source code history searching for possible rejuvenation scenarios that we further validate, via a manual analysis, focusing on commits that introduce many occurrences of *auto-typed variables*, *lambda expressions*, and *range-based for*. Second, we contacted KDE developers that implemented these rejuvenation efforts in their programs to answer the following questions:

SQ1 *What are the main motivations for adopting modern C++ features, such as lambda expressions, range-for, and declarations using the auto operator?*

SQ2 *How often do you conduct software rejuvenation efforts? Do you use any tool to support this kind of software maintenance?*

SQ3 *Is there any direction from the KDE community about when features of a new version of the C++ language should be used more widely in KDE projects, or is this an individual or small team decision in each project?*

SQ4 *What are the main reasons for the adoption rate of modern C++ features having increased substantially between 2015 and 2016?*

SQ5 *We have found that some features for concurrent programming in modern C++ (like thread declaration, futures, and async) are rarely used within the KDE projects. Is there an explanation for this?*

The first strategy (mine the source code history) allowed us to answer (RQ7). In the second strategy (survey KDE developers), we use the answers from question (SQ1) of the questionnaire to address our research question (RQ6), while question (SQ2) allows us to tackle question (RQ5). Furthermore, questions (SQ1), (SQ3), and (SQ4) give us insights into the motivations behind the trends, allowing us to explore (RQ3) in more detail. Finally, (SQ5) helps us understand the motivations that lead KDE developers not to use some modern C++ features like async and concurrence. We surveyed KDE developers primarily through email correspondence and conducted one interview with a contributor through an online meeting to collect data. We consolidated the survey results and presented a summarized overview of the findings using quoted text.

5.1 | Rejuvenation Efforts

In Sections 3 we detail our conservative approach for mining rejuvenation efforts: source code patches that increase by at least 50% the amount of a modern C++ language feature (such as *auto-typed variables*, *lambda expressions*, *range-based for*) and do not increase the total number of project statements in more than 5%. This conservative approach might lead to both false positives and false negatives. The results of our conservative approach to identifying rejuvenation efforts reveal a total of 81 commit candidates, from which we were able to manually confirm that 57 of them (70.37%) correspond to true positives—that is, commits that update the code with the sole goal of rejuvenating a program. More interesting, part of large rejuvenation efforts (8.7%) were conducted with the support of automatic refactoring tools.

These 57 rejuvenation efforts we confirm relate to 43 distinct projects in our dataset (15.80% of 272 projects). Also, of the 57 rejuvenation commits found, seven commits correspond to the introduction of *lambda expressions*, 27 commits correspond to the introduction of *auto-typed variables*, and 27 commits correspond to the introduction of *range-based for*. We found commits that introduce more than one feature, though. For instance, commit ea924b146 of `plasma-framework` (commit date on 7 May 2015 and message *port libplasma away from sycoca as much as possible*) introduces *lambda expressions*, *auto-typed variables*, and *range-based for*. Other examples include:

- Commit 2e508ac of `kdenlive` (commit date on 20 Apr 2017 and message *Use modern `for`*) replaces more than 200 occurrences of the `foreach` Qt statement[7] by the new *range-based for* statement in 59 files. Figure 4 shows one example of the changes in this particular commit. Similar to several other contributions, the sole purpose of this commit was to rejuvenate the source code to use modern C++ features.

---

[7]https://doc.qt.io/qt-6/foreach-keyword.html

- Commit 5c28a3e5 of `labplot` (commit date on 3 Dec 2017 and message *Replace foreach macros with range-based for loops in many places.*) is another example of a large transformation that replaces more than 300 occurrences of the `foreach` macro by the modern *range-based for* statement. This particular commit changes a total of 36 files. Listing 2 in Figure 4 shows an example of transformation in this particular commit—which also introduces more than 250 new declarations using the *auto-typed variables* feature for type deduction.

- Commit f519ce07 of `kid3` (commit date on 29 Sep 2018 and message *Use auto where it improves the readability.*) corresponds to another example of code rejuvenation, introducing more than 400 *auto-typed variables* in 88 files. This commit is also representative w.r.t the KDE developer's adoption of tools to support code rejuvenation efforts. In the full message, the author says *Refactored automatically using clang-tidy.* See an example in Listing 3 (Figure 5). Listing 4 in Figure 5 shows an example of transformation in this particular commit. Another commit 7e6ff7d7 of `akregator` (commit date on 2 Nov 2020 and message *Modernize code*) introducing more than 94 *auto-typed variables* in 41 files.

```
if (!pattern.isEmpty()) {
  QVariantList mapped;
- foreach (const QModelIndex &ix, m_model->getChildrenIndexes()) {
+ for (const QModelIndex &ix : m_model->getChildrenIndexes()) {
    mapped << m_proxyModel->mapFromSource(ix);
  }
  // ...
}
```

Listing 1 Example of introducing *range-based for* in a large commit from `kdenlive`

```
- foreach (PluginLoader *loader, m_pluginsWithErrors) {
+ for (auto* loader : m_pluginsWithErrors) {
      if (loader->fileName() == absolutePath) {
          m_pluginsWithErrors.removeAll(loader);
          delete loader;
          break;
      }
  }
```

Listing 2 Example of introducing *range-based for* in a large commit from `labplot`

FIGURE 4 Example of introducing *range-based for*

Although we found several commits that introduce many occurrence of *auto-typed variables* and *range-based for*, this kind of rejuvenation effort is more scarse for *lambda expressions*. Commit 40a2aee94 of `kbibtex` (commit date on 20 Jul 2019 and message *Refactoring usage of QSignalMapper by using lambda functions in QObject::connect calls.*) introduces new 40 *lambda expressions* distributed in 16 files. Listing 5 in Figure 6 shows one example of the changes in this particular commit. Finally, commit 034e0b7b of `kwidgetsaddons` (commit date on 7 May 2021 and message *Modernise code base...Less SLOT() usage where PMF or lambda works*) introduce more than 50 new *lambda expressions* distributed in 65 files. Listing 6 in Figure 6 shows one example of the changes in this particular commit.

> **RQ4: Do KDE developers conduct maintenance efforts having the sole goal of rejuvenating C++ code?**
>
> We found evidence that KDE developers conduct rejuvenation efforts to adopt *auto-typed variables*, *lambda expressions*, and *range-based for*—maintenance tasks whose goal was mainly to refactor the code to replace legacy features with new ones. Even considering an overly conservative approach, we identified more than 50 rejuvenation efforts, changing several hundreds of lines of code to modernize C++ code.

```
while (listIndex < completions.size()) {
  const QByteArray& name = completions.at(listIndex++);
  if (name.left(textLen) == text) {
- char* r = reinterpret_cast<char*>(::malloc(name.length() + 1));
+ auto r = reinterpret_cast<char*>(::malloc(name.length() + 1));
    qstrcpy(r, name.constData());
    return r;
  }
}
```

Listing 3 Example of introducing *auto-typed variables* in a large commit from kid3

```
static DeleteSubscriptionJob *reallyCreateJob(TreeNode *node)
{
- DeleteSubscriptionJob *job = new DeleteSubscriptionJob;
+ auto *job = new DeleteSubscriptionJob;
    job->setSubscriptionId(node->id());
    return job;
}
```

Listing 4 Example of introducing *auto-typed variables* in a large commit from akregator

FIGURE 5 Example of introducing *auto-typed variables*

```
for (int i = 0; i < ordinals.length(); ++i) {
- QAction *editionAction = editionMenu->addAction(ordinals[i], sm, SLOT(map()));
- sm->setMapping(editionAction, i + 1);
+ QAction *editionAction = editionMenu->addAction(ordinals[i]);
+ connect(editionAction, &QAction::triggered, p, [this, i]() {
+ setEdition(i + 1);
+ });
}
```

Listing 5 Example of introducing *lambda expressions* in a large commit from kbibtex

```
@@ -157,8 +157,12 @@ void KAnimatedButtonPrivate::updateIcons()
     newMovie = new QMovie(icon_path);
     frames = 0;
     newMovie->setCacheMode(QMovie::CacheAll);
- QObject::connect(newMovie, SIGNAL(frameChanged(int)), q, SLOT(_k_movieFrameChanged(int)));
- QObject::connect(newMovie, SIGNAL(finished()), q, SLOT(_k_movieFinished()));
+ QObject::connect(newMovie, &QMovie::frameChanged, q, [this](int value) {
+ movieFrameChanged(value);
+ });
+ QObject::connect(newMovie, &QMovie::finished, q, [this] {
+ movieFinished();
+ });
```

Listing 6 Example of introducing *lambda expressions* in a large commit from kwidgetsaddons

FIGURE 6 Example of introducing *lambda expressions*

5.2 | Responses from KDE developers

We identified thirty-two developers from the KDE community who were responsible for driving the 57 commits that characterize the rejuvenation efforts according to our heuristics. We contacted these developers to understand the motivations that led them to rejuvenate the project's source code and received responses from 34.37% of the developers contacted. We used the TF (Truck Factor) metaphor (Section 3.2) and found that 7 (63.63%) of the 11 developers who responded to our contact are in the list of top developers of the KDE projects in which they conducted rejuvenation efforts. Also, top developers

were responsible for 36 (63.2%) of the rejuvenation commits found by our heuristic. Table 2 presents the characteristics of the developers participating in our survey. The first column presents the identification of the participant in our study. The second column contains the project name on which this developer performed a modernization effort. The third column presents the number of contributions of the developer to the project. The fourth and last columns inform whether this developer is a Core Developer according to the TF metaphor.

> **RQ7: Are the Core Developers of the projects responsible for conducting rejuvenation efforts in KDE projects?**
>
> We used the TF metaphor and found that 36 (63.2%) of the 57 commits marking the largest rejuvenation efforts were made by the core developers of the project. Our results suggest that software rejuvenation is a high-level concern that requires the involvement of technical leaders to conduct large maintenance efforts.

TABLE 2 Group of KDE developers that participated in the survey and conducted rejuvenation efforts.

Id (#)	Project Name	Commits realized (#)	Is Core Developer?
P1	kphotoalbum	1602	Yes
P2	kwin	2473	Yes
P3	calligra	271	No
P4	partitionmanager	495	Yes
P5	akonadi	1688	Yes
P6	discover	6	No
P7	labplot	3801	Yes
P8	plasma-framework	2	Yes
P9	kdevelop	2133	Yes
P10	amarok	14	No
P11	okular	47	No

Our data analysis consists of a process involving annotations extracted from the developers' responses following a two-step coding approach: the first consisted of the open coding process followed by a second step to categorize the codes according to the recommendations described by Saldaña's [35]. The first author conducted the coding under the supervision of the fourth author, who supported code validation and categorization. This process resulted in 7 main categories and 35 codes. The followings are the categories and the codes associated with them.

- **Motivations to rejuvenate**: Code quality improvements, Avoid writing boilerplate code, Performance improvements, Modern features can reduce error-prone, Code rejuvenation can attrack new contributors, Modern features allow faster writing of code, Delays Software aging, Prevent the software from dying.

- **When Rejuvenations Occur**: When the benefits of modern features are considerable, When the code really needs to be rewritten, Incremental code rejuvenation for small changes, When the compiler supports the modern features, When the frameworks evolves, Rejuvenation efforts are conducted during bug fixing, During the implementation of modern C++ features, Developers conduct rejuvenation efforts when learning about the modern C++ features, Rejuvenation efforts are made when the quality of the code has already degraded.

- **Tooling support**: Tooling support to identify rejuvenation opportunities, Tooling support for large rejuvenation efforts, Automated code rejuvenation can introduce bugs, Tooling support can reduce rejuvenation costs.

- **Challenges**: High cost of making changes, Incompatibility of modern features with old versions of frameworks, Habit of using existing features makes it difficult to adopt modern features, Modern features may worsen readability,modern features that require a change in logic are harder to port automatically.

- **Existence of third-party libraries**: Existing features facilitate the work, Existing features have better integration with APIs from the same library, The features made available by third-party libraries meet expectations.

- **Decision to rejuvenate**: The project team decides when to rejuvenate, Individual members can decide to perform rejuvenation.

- **Direction for rejuvenation**: Framework libraries influence rejuvenation, The compiler influences the language version to use, Discussions of developers community influences the adoption of modern features, the community decides which compiler version to use.

5.2.1 | Motivations to rejuvenate

This category summarizes the motivations and factors found in this research that drive developers in the KDE community to rejuvenate their programs. The most emerging motivations pointed out by the participants are related to the benefits acquired by adopting the modern features of the C++ language in their software. All participants report on code quality improvements (increased code understanding, making code more concise, readability improvements, ease of maintenance activities, security improvements, and machine code quality improvements).

> My personal motivations for adopting new C++ features has been to write code which is easier to understand and more concise, as well as the convenience that these new features bring.
>
> *Kphotoalbum - P1*

> Lambda expressions come very handy in Qt's signal and slot connections, the code is more compact in such places...range based for-loops also allow for a more compact code. Same for the auto- keyword, especially when dealing with enums inside of class namespaces or so - this is where the auto-keyword can save a lot of space and a lot of typing work.
>
> *Labplot - P7*

Furthermore, participants (P1, P3, P6-P10) report that the adoption of modern features like *lambda expressions*, *range-based for*, and *auto-typed variables* reduces the amount of boilerplate code and makes the code leaner. Another benefit reported by participant P3 was performance gain when adopting *range-based for*. Participants P5 and P10 report that they performed rejuvenation to attract new collaborators to the project. Participant P5 reports that the use of modern features makes programs less error-prone.

> A big factor is to reduce boilerplate code, to get the code much more readable (for instance range for instead of plain iterators.. at least where possible) speaking about iterators, also something like **auto it = myLisy.begin**(); is much more readable than **QList < QString >:: iterator ii = myList.begin**();.
>
> *Plasma framework - P8*

> Range-for was already kind of present through some Qt macro kung-fu, so it's more getting rid of this magic to switch to the proper way of doing things, and getting some performance improvements by removing useless copy constructions…
>
> *Calligra - P3*

> My main motivation is better legibility and more concise and expressive code. I believe all three lead to code that is easier to maintain and less error-prone.
>
> *Akonadi - P5*

Finally, participant P10 reports that software rejuvenation must be performed periodically to slow down software aging and increase its lifespan, otherwise, the software may die.

> In the case of the linux client here at the company, I have been doing it weekly, because like, here I have been using GTK instead of Qt, but it is the same. Because GTK has evolved, the language has evolved and I want the client to be more readable, I do it weekly. In the end, if you don't do this, nobody will be interested anymore, the software will get old and will die....so this work has to be done periodically or it will die.
>
> *Amarok - P10*

> **RQ6:** What are the reasons that motivate KDE developers to conduct maintenance efforts for code rejuvenation?
>
> Our results show that KDE developers perceive that modern C++ features can improve code readability, reduce the error-prones, and simplify software maintenance. They use ***lambda expressions, range-based for***, and ***auto-typed variables*** to eliminate boilerplate code by avoiding creating unnecessary classes, rewriting variables with long names, and making the code cleaner. In addition, developers rejuvenate the code base of their programs in an attempt to attract new contributors to the repository, as well as delay aging and prevent software from reaching the end of its useful life. Such reported benefits show some advantages in using modern C++ features. Also, KDE developers rejuvenate their programs' source code with the goal of attracting new contributors to the project.

5.2.2 | When Rejuvenations Occur

This category summarizes the situations, moments, and ways in which some of the rejuvenation efforts reported by developers in the KDE community are taking place. Developers P3, P8, and P10 reported making rejuvenation efforts as the framework evolves. Other rejuvenation efforts take place when the compiler supports modern features. According to developers P1 and P7, rejuvenation occurs gradually as small changes are made to the C++.

> " It depends on the project mainly. Most of the time, a major Qt upgrade (Qt4=>Qt5, Qt5=>Qt6) will require a more modern C++, thus allowing the full feature set. But there are some projects that will simply use these features when they consider most distributions will have a modern compiler handling them. "
>
> *Calligra - P3*

Some developers report some conditions that need to be evaluated to decide whether rejuvenation should be performed. Participant P1 and P8 perform rejuvenation only when the program code needs to be rewritten. Participant P1 reports that rejuvenation is always performed when the benefits of introducing a modern C++ feature are significant. Participant P9 reports that rejuvenation is performed when code quality has already deteriorated.

> " it's more continuous and low intensity, new code tends to use more modern C++ features, old code is ported more when it really needs to be rewritten. "
>
> *Plasma-framework - P8*

KDE developers report that rejuvenation efforts occur during code creation and review. Participant P5 reports that he takes rejuvenation actions when he fixes bugs in the program. On the other hand, rejuvenation efforts may also take place after a new feature is implemented, according to Participants P4 and P5. Finally, participant P6 reports that some rejuvenation efforts are made when he learns the modern features of the C++ language.

> " Often code modernization just happens by writing new modern code. Occasionally when doing major porting, e.g, Qt4 to Qt5, or Qt5 to Qt6. "
>
> *Partitionmanager - P4*

> " I usually end up modernizing a particular piece of code when I'm fixing some bug or implementing a feature there and I notice that the particular area of code base could benefit from a little bit of modernization. "
>
> *Akonadi - P5*

KDE developers report various situations in which they perform rejuvenation efforts, such as when enhancing frameworks, performing code reviews to fix bugs, and implementing new features into the program. There are some criteria they use to decide whether or not to perform rejuvenation efforts, such as whether the compiler now supports modern language features. In addition, developers report that program rejuvenation is performed incrementally for small changes to the language.

5.2.3 | Tooling support

This category summarizes KDE developers' perceptions of tooling support to advance rejuvenation efforts in their programs. Most of the developers (81.8%) surveyed report using some tool to support rejuvenation efforts. Participants P3, P7, and P10

report using automated tools to identify opportunities for rejuvenation. Participant P1 reports that automated tools can be useful when major rejuvenation efforts are needed. In addition, participant P9 reports that tool support can reduce the cost of rejuvenation. The participants P3. P6 and P9 report that automated tools (like Clazy, and Clang-tidy) have adequate tooling support for process rejuvenation efforts automatically. Finally, the automated tools to support program rejuvenation used by developers in the KDE community are Clazy [8], Clang-tidy [9], KDevelop [10], and Clion [11]

> We had some phases were we spent dedicated time on our side to modernize the code a bit. We used clang-tidy to identify and to adjust the code ... more was done later manually in multiple further iterations from time to time. Right now these activities are more or less done, we follow our code style using the C++11 features and there is no need anymore for a "modernization effort".
>
> *Labplot - P7*

> Depending on the feature. C++11 was a huge improvement over C++98, which meant that a dedicated modernization effort was warranted. Tools that we used at this time were refactoring scripts ... written by other KDE developers.
>
> *Kphotoalbum - P1*

> On the tooling side, we are pretty well equipped, with clang-tidy and clazy. I use both regularly, and I think the same is true for many other KDE community members. They are able to modernize some patterns automatically. clang-tidy is useful for general C++ modernization, while clazy handles Qt specific things.
>
> *Discover - P6*

> Clazy introduced checks that suggested rewriting code to use several C++11 features. If you look at the history of the $foreach$ check (it suggests replacing an old $Q\_FOREACH$ macro with range-based for), you can see it was created back in 2015. [...] Rewriting code to use auto or range-based for is simple and can be helped by tools (and I don't know about the other devs, but most of my for loops use auto variables, the two go together nicely). Using lambda, on the other hand, is something you will do when writing of fixing code for other purpose, not because a tool told you could use one there, but because you see a pattern where lambdas will be helpful.
>
> *Calligra - P3*

RQ5: Which tools do KDE developers use to support maintenance efforts for code rejuvenation?

Our findings show that KDE developers use tools (Clazy, Clang-tidy, KDevelop, and Clion) to support rejuvenation efforts. According to their reports, they mainly use the tools to identify scenarios that can be rejuvenated. In addition, automated tools can support major rejuvenation efforts and reduce the costs of these changes according to reports from developers in the KDE community. Finally, developers reported that features such as *range-based for* and *auto-typed variables* have adequate tooling support for process automation, while *lambda expressions* require additional cognitive effort from developers to be included in a code snippet of a program.

5.2.4 | Challenges

This category summarizes some factors that KDE developers believe can prevent programs from being rejuvenated. Most developers report at least one factor that can prevent or delay program rejuvenation. Participants P6, P8, and P10 report that some of the modern features can affect code readability. Developers P3 and P6 report that some features are not adopted due to incompatibility with older versions of the framework. In addition, participants P1 and P9 report that the cost of performing rejuvenation is high. Finally, participants P3 and P4 report that the habit of using legacy features of the C++ language can delay the adoption of modern C++ features. Finally, participant P5 reports that features that require logic changes are difficult to port automatically.

---

[8]Clazy, is a static code analyzer based on the Clang framework.
[9]Clang-tidy is a clang-based C++ linter tool.
[10]KDevelop is an extensible IDE plugin for C/C++ and other programming languages.
[11]Clion CLion is a C/C++ development environment with many built-in features.

> So in a business context it's relatively difficult to get time/ budget for doing these kind of things, usually it's being done when the code quality has degraded quite a bit already, i.e. when it's too late or people finally get too annoyed working with the legacy code base.
>
> *Kdevelop - P9*

> But some features have been integrated in Qt for a very long time, and the newcomers have a hard time rising above and replacing years of habits. For instance, std::thread. It's a welcome addition to the C++ standard library, and it was badly needed. But the need for an abstraction to system threading APIs is not new, and for a long time Qt had a QThread class, deeply rooted in its object and event models.
>
> *Calligra - P3*

> Mostly, Qt already has facilities for concurrent programming. Moving from these to different ones has a far higher cost as with the features mentioned in the first question. I would therefore expect the uptake to be slightly higher in newly written applications.
>
> *Kphotoalbum - P1*

> With auto, especially when using templated classes, writing new code is much more comfortable, since long type signatures don't need to be spelled out every time. Also, when changing the type of a variable, with auto the amount of code that needs to be changed becomes a lot less. However, it might even sacrifice some of the readability to speed of development. At least while using IDEs, this isn't a problem in practice.
>
> *Discover - P6*

> When KDE adopted C++11 we used some scripts to automatically port from the old connect() syntax to the new syntax [0], although regarding latest C++17 features (e.g. optional), those are hard to port automatically as usually change in code logic is needed.
>
> *Akonadi - P5*

KDE developers face some challenges in rejuvenating their programs, such as the incompatibility of legacy versions of the framework, and projects often need to maintain compatibility with older platforms that do not yet support the modern C++ features. Some features such as **auto-typed variables** can affect the readability of the code. Moreover, the habit of using legacy features may delay the adoption of modern features by developers. Another problem for developers is the high cost of program changes. Finally, some features require cognitive efforts to be introduced in the code and are difficult to port automatically.

5.2.5 | Existence of third-party libraries

This category summarizes the motivations that lead KDE developers to use existing features instead of modern features of the C++ language. In our quantitative analysis, we found that modern features for working with multithreading are rarely used by KDE developers. Some features that are not included in the standard library are available in third-party libraries (Qt) that facilitate developers' work. Participants P2, P3, P5, and P6 report that C++ language features provided by third-party libraries integrate better with developed APIs that support the same library. Participants P1, P7, and P10 report that these features meet the current needs of developers and that they do not need to make changes to adopt modern features immediately.

> This is available in Qt already for very long time and many projects, including LabPlot and Cantor, are using this. Right now there is no need and no plans on our side to move away from what Qt is offering. This can change in future, of course, but right now our spare time is better invested in other areas.
>
> *LabPlot - P7*

> Such preference is motivated by the fact that QThread is integrated with other Qt APIs better than say std::thread.
>
> *Kwin - P2*

> Multi-threading is mostly done by the means of QThread and QObjects. Methods of QObjects can be executed indirectly by the Qt metaobject system, and that is clever enough to call functions from the QThread which the object belongs to. For futures and async, there is a Qt framework (QtConcurrent) which provides similar APIs that are easier to integrate into existing Qt projects.
>
> *Discover - P6*

Our results show that KDE developers use the modern C++ features provided by the Qt library to work with threads and that these language features meet the current needs of developers. Moreover, the feature provided by the Qt library is easier to integrate into the APIs of their projects. According to these developers, there is no need for a migration to use the new library currently available in the standard library.

5.2.6 | Decision to rejuvenate

In this category you will find those responsible for performing rejuvenation, i.e. those who make the decision to rejuvenate a program by the developers of the KDE community. Most participants (P1-P6 and P9) report that the decision to rejuvenate is made by the development team. Participants P5-P7 and P10 report that this decision can also be made by an individual member.

> This is a team decision in each project. Of course, one looks at other projects (especially the frameworks libraries) as a reference, but ultimately every team can make their own rules.
>
> *Kphotoalbum - P1*

> The KDE Community is working on multiple bigger project like the Frameworks, Plasma but also such big applications like Krita, Kdenlive, LabPlot, Cantor, etc. There are code styles and minimal compiler requirements within every single project and there is no "global" prescription for what to use. The project maintainers decide on their own what they need and when. I'm mainly working on LabPlot and Cantor and we have C++11 as the minimum supported version now.
>
> *Labplot - P7*

The results presented show that the decision to rejuvenate a program in the KDE community is made by the development team and can also come from a single team member.

5.2.7 | Direction for rejuvenation

This category summarizes the factors that influence the use of modern features of the C++ language. Developers report that there is no direction from the community, but some factors may influence the decision of developers in the KDE community developers. Participants P1, P3, P4, and P5 report that framework libraries, when using modern C++ features, influence rejuvenation. Participants P6, P9, and P10 report that they can be influenced to use modern C++ features by following mailing list discussions, talks, and community events and C++ language. In addition, the community decides which compiler version to use, which in turn influences which language version to use and which features are supported by the compiler.

> But the KDE Frameworks will have a bigger influence, and they are much more carefully driven than the applications. If one of the major framework was to require, say, coroutines (in the future obviously), it sure would influence the applications into using this feature even for things not related to the said framework.
>
> *Calligra - P3*

> The standard set in KDECompilerSettings.cmake is updated every now and then, after some discussion on some KDE mailing list (kde-devel, kde-core-devel) weighting the pros and cons.
>
> *Kdevelop - P9*

> Usually there is no organized modernization effort in KDE. People tend to start to modernize their own projects when they learn about the new possibilities, and if I remember correctly there were modern C++ talks at Akademy, which certainly helped adoption.
>
> *Discover - P6*

> ❝ Nowdays the modernization effor on C++ go a bit hand in hand with another effort as we are in the middle of the transition between Qt5 and Qt6, which will also bring a major, source and binary incompatible major release of the KDE frameworks, that leads to a general modernization effort. ❞
>
> *Plasma-framework - P8*

Our fidings suggest that KDE developers report that the community does not provide direction, but their participation in discussion lists, talks, and events can influence the decision to rejuvenate a program. In addition, the evolution of the frameworks influences the decision of developers in the KDE community to use modern C++ features.

5.3 | The primary reasons for the increased adoption of modern C++ features

As presented in Section 4, we found a trend (see Figure 3) of adoption for three modern C++ features (*auto-typed variables*, *lambda expressions*, and *range-based for*) that had increased substantially between 2015 and 2016. This trend motivated us to investigate and additional question: **What are the main reasons for the adoption rate of modern C++ features having increased substantially in this period?** Based on the responses of the interviewed developers and our findings (Sections 4, 5.1, and 5.2), three main reasons might explain that trend:

*R*1 C++ projects within the KDE community are closely linked to the Qt framework. In 2016, version 5.7 [12] of the framework was released, which requires compilers to support the C++11 version (like Clang, GCC). According to our interviewees (P3, P4, and P8), during this period, developers began migrating to the Qt5 version of the framework and occasionally started adopting the modern C++ features of the language made available by the C++11 standard. Additionally, 55 (96.49%) commits that characterizes rejuvenation efforts identified by our heuristic, were made between 2016-2022. See quotes below with the developers' responses.

> ❝ It depends on the project mainly. Most of the time, a major Qt upgrade (Qt4=>Qt5, Qt5=>Qt6) will require a more modern C++, thus allowing the full feature set. But there are some projects that will simply use these features when they consider most distributions will have a modern compiler handling them. A lot of KDE projects will use tools like Clazy and Clang-tidy to help identify parts of code that need modernizing. Clazy is really Qt oriented while, as you surely know, Clang-tidy is a generic tool. ❞
>
> *Calligra - P3*

> ❝ It's mostly individual decisions. However, there is also baseline determined by Qt requirements. E.g. Qt5 needs at least C++11 and Qt6 needs C++17, so all KDE software can depend at least on C++11 or C++17 if it targets Qt6. ❞
>
> *Partitionmanager - P4*

*R*2 In 2015, the Clazy tool introduced changes [13] that suggest the use of modern features in the C++11 standard. Additionally, KDE developers (participants P3, P6, and P9) reported that features such as range-based for and auto have adequate tooling support for process automation (e.g., replacing a deprecated Q_FOREACH macro), while lambda expressions require additional cognitive effort from developers to be included in a code snippet of a program. See quote below.

> ❝ Clazy introduced checks that suggested rewriting code to use several C++11 features. If you look at the history of the *foreach* check (it suggests replacing an old $Q\_FOREACH$ macro with range-based for), you can see it was created back in 2015. [...] Rewriting code to use auto or range-based for is simple and can be helped by tools (and I don't know about the other devs, but most of my for loops use auto variables, the two go together nicely). Using lambda, on the other hand, is something you will do when writing of fixing code for other purpose, not because a tool told you could use one there, but because you see a pattern where lambdas will be helpful. ❞
>
> *Calligra - P3*

---

[12] https://wiki.qt.io/New_Features_in_Qt_5.7
[13] https://github.com/KDE/clazy/commits/master/src/checks/level1/foreach.cpp

*R*3 According to P3 and P8, the KDE projects often need to maintain compatibility with older platforms that do not yet support the new C++ language constructions. This could explain why adoption slowed between 2012 and 2014. See quotes below with the developers' responses.

“ One first big factor is that some projects can't adopt modern C++ that fast perhaps because they need to support old platforms where that is not possible (think about a commercial Windows app that perhaps still supported XP until a couple of years ago, or very old embedded systems..) We don't have this requirement as our target platform is primary linux/bsd distributions in their current release (and secondary android,windows and mac, but also for those, only recent releases). ”

Plasma-framework - P8

“ There is no major direction given. Each project does as it wants. For instance, calligra chose for a long time to keep compatibility with some very old libraries because one of our known users, Jolla, is still using Qt 5.6, prohibiting some new constructions from being used. But the KDE Frameworks will have a bigger influence, and they are much more carefully driven than the applications. If one of the major framework was to require, say, coroutines (in the future obviously), it sure would influence the applications into using this feature even for things not related to the said framework. ”

Calligra - P3

6 | DISCUSSION

In this section we discuss the implications of our results (Section 6.1), and present some limitations that might threat the validity of our work (Section 6.2).

6.1 | Implications of the Results

We conducted a large-scale study of C++ projects in the KDE community. We evaluated the use of modern features such as *auto-typed variables, lambda expressions, range-based for* in community projects and contacted developers who have made efforts to rejuvenate their programs. In this section we summarize the implications of our study on some topics:

- We present a list of benefits and rejuvenation scenarios that developers can adopt in their programs. Our results for **RQ1, RQ3, RQ4, and RQ6** show that using ***lambda expressions, range-based for***, and ***auto-typed variables*** provides benefits such as readability, makes code cleaner, and simplifies software maintenance. KDE developers also report that using *lambda expressions* reduces the amount of boilerplate. Lucas et al[19,36] show that Developers also realize these benefits of using *lambda expressions* in Java programs. Our findings for **RQ2** show that the use of *lambda expressions* is recommended by KDE developers in conjunction with resources for working with Qt's signals and slot connections in C++ programs.

- Our findings for **RQ5** demonstrate the use of automated tools (Clazy, Clang-tidy, KDevelop, and Clion) to help KDE developers identify rejuvenation scenarios. In addition, the tools can reduce the cost of rejuvenation efforts, which benefits developers in terms of the time it takes to complete these efforts. A broader audience of C++ developers can also benefit from these findings.

- Our results for **RQ4** present a catalog of commits 1 that characterizes rejuvenation efforts. We hope the examples in the catalog could help software developers to rejuvenate their programs and tool builders to identify opportunities to implement source code transformations.

- Our findings for **RQ7** revealed that core developers were responsible for most of the rejuvenation commits found to adopt modern C++ features in our dataset. It's similar to the occurrences reported by Mazinanian et al.[5] for Java Developers.

- KDE developers report that *lambda expressions* requires additional cognitive effort from developers to be included in a code snippet of a program. According to Uesbeck et al.[23] using *lambda expressions* negatively affects productivity for less

experienced developers regarding how quickly they can write correct programs. Based on these results, we recommend that rejuvenation efforts to include *lambda expressions* be undertaken by experienced developers with knowledge of the codebase of programs.

6.2 | Threats to Validity

Regarding **construct validity**, we have employed the Truck Factor (TF) metaphor[28] to identify the core developers of KDE projects. Our research findings indicate that the core developers contributed to 63.15% of the rejuvenation efforts identified through our conservative approach. Critics may question our decision to use TF as a proxy for identifying the core developers, but previous studies[30] have demonstrated the effectiveness of this metric. Furthermore, we assert that other metrics[29] could have yielded the same result, which is that technical leaders are responsible for significant rejuvenation efforts. Additionally, this result is consistent with our qualitative assessment, as seven out of eleven survey respondents were core developers in the projects.

The accuracy of our conservative approach to identifying rejuvenation efforts represents an **internal threat** for our research. In section 5.1, we show that our conservative approach resulted in 57 (70.37%) of the rejuvenation efforts between 81 intervals we collect. We stressed that our initial research goal was to evaluate whether the KDE community developers conducted large rejuvenation efforts. To achieve this goal, we would not necessarily have to find all the contributions that characterize rejuvenation efforts, but some that showed the occurrence of this phenomenon in the KDE community projects. That is, here we privilege precision over recall, and many other rejuvenation efforts might not have been captured using our conservative approach.

In terms of **external validity**, we analyzed 272 out of 1050 C++ projects from the KDE community. The projects we selected satisfy the criteria of having more than ten years of development and at least one recent contribution (after January 01, 2021). This represents a wide range of application domains so that we can generalize our results to the KDE organization as a whole. However, studies in source code repositories from other organizations might reveal different results. Furthermore, we cannot generalize our findings to closed-source projects because we only look at open-source repositories. Besides that, we believe that developers in general can benefit from the rejuvenation practices we present in our study. Also related to **external validity**, we contacted 32 KDE developers that led the rejuvenation efforts found using our heuristic, to better understand their motivations to use modern C++ features. Eleven developers (34.37% of the initial population) answered our survey, which does not allow us to generalize our findings.

7 | CONCLUSIONS

During this research, we investigate the practices C++ developers use to rejuvenate open-source programs in the KDE organization. Our findings bring evidence about a trend toward the widespread adoption of modern C++ features, including *lambda expressions*, *range-based for*, and *auto-typed variables*. Nonetheless, to our surprise, the new concurrent support for C++ multithreading is rarely used in KDE projects. The main reason for this observation is the use of the Qt support for multithreading in KDE projects, which is not being replaced by the new standard multithreading features of C++.

Our research also revealed large modernization efforts, some of them replacing hundreds of occurrences of legacy constructs with the *range-based for* and *auto-typed variables* statements, for instance. Some of these efforts have even been conducted using the support of tools, such as Clazy and Clang-Tidy. After surveying KDE developers responsible for these large modernization efforts, we identified a couple of reasons that motivate this kind of software maintenance, which aims at improving the readability, conciseness, and expressiveness of the code, slowing software aging, and even attracting new contributors to the project repositories. We believe our findings could help a broader range of C++ developers in taking a decision on whether or not to rejuvenate their programs.

Our research findings indicate that developers perceive some benefits to adopting modern features of the C++ programming language and how they adopt them in their programs. Programming language designers need to know how developers embrace the new features of the language and the impact on their programming productivity. We presume that this is an important aspect that merits a thorough investigation. For future work, we advise conducting exploratory research focused on the interests of software language designers. We also intend to look into the technical aspects to suggest improvements for automated tools and help developers to rejuvenate their software.

ACKNOWLEDGMENTS

References

1. Bjarne Stroustrup. Thriving in a crowded and changing world: C++ 2006-2020. *Proc. ACM Program. Lang.*, 4(HOPL): 70:1–70:168, 2020. . URL https://doi.org/10.1145/3386320.

2. Jeffrey L. Overbey and Ralph E. Johnson. Regrowing a language: refactoring tools allow programming languages to evolve. In Shail Arora and Gary T. Leavens, editors, *Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009, October 25-29, 2009, Orlando, Florida, USA*, pages 493–502. ACM, 2009. . URL https://doi.org/10.1145/1640089.1640127.

3. Santiago Bragagnolo, Nicolas Anquetil, Stéphane Ducasse, Abderrahmane Seriai, and Mustapha Derras. Software Migration: A Theoretical Framework (A Grounded Theory approach on Systematic Literature Review). Research report, Inria Lille Nord Europe - Laboratoire CRIStAL - Université de Lille, 2021. URL https://hal.inria.fr/hal-03171124.

4. Anwar Alqaimi, Patanamon Thongtanunam, and Christoph Treude. Automatically generating documentation for lambda expressions in java. In Margaret-Anne D. Storey, Bram Adams, and Sonia Haiduc, editors, *Proceedings of the 16th International Conference on Mining Software Repositories, MSR 2019, 26-27 May 2019, Montreal, Canada*, pages 310–320. IEEE / ACM, 2019. . URL https://doi.org/10.1109/MSR.2019.00057.

5. Davood Mazinanian, Ameya Ketkar, Nikolaos Tsantalis, and Danny Dig. Understanding the use of lambda expressions in java. *Proc. ACM Program. Lang.*, 1(OOPSLA):85:1–85:31, 2017. . URL https://doi.org/10.1145/3133909.

6. Aditya Kumar, Andrew Sutton, and Bjarne Stroustrup. Rejuvenating C++ programs through demacrofication. In *28th IEEE International Conference on Software Maintenance, ICSM 2012, Trento, Italy, September 23-28, 2012*, pages 98–107. IEEE Computer Society, 2012. . URL https://doi.org/10.1109/ICSM.2012.6405259.

7. Peter Pirkelbauer, Damian Dechev, and Bjarne Stroustrup. Source code rejuvenation is not refactoring. In Jan van Leeuwen, Anca Muscholl, David Peleg, Jaroslav Pokorný, and Bernhard Rumpe, editors, *SOFSEM 2010: Theory and Practice of Computer Science, 36th Conference on Current Trends in Theory and Practice of Computer Science, Spindleruv Mlýn, Czech Republic, January 23-29, 2010. Proceedings*, volume 5901 of *Lecture Notes in Computer Science*, pages 639–650. Springer, 2010. . URL https://doi.org/10.1007/978-3-642-11266-9_53.

8. Bjarne Stroustrup. Evolving a language in and for the real world: C++ 1991-2006. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, HOPL III, page 4–1–4–59, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 9781595937667. . URL https://doi.org/10.1145/1238844.1238848.

9. Alexander Stepanov and Meng Lee. *The standard template library*, volume 1501. Hewlett Packard Laboratories 1501 Page Mill Road, Palo Alto, CA 94304, 1995.

10. Bjarne Stroustrup. *The C++ Programming Language, Third Edition*. Addison-Wesley Longman Publishing Co., Inc., USA, 3rd edition, 1997. ISBN 0201889544.

11. N.M. Josuttis. *C++ 17: The Complete Guide*. Nicolai Josuttis, 2019. ISBN 9783967300178. URL https://books.google.com.br/books?id=UAmQzQEACAAJ.

12. Scott Meyers. *Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14*. O'Reilly Media, Inc., 1st edition, 2014. ISBN 1491903996.

13. Boris Schäling. *The boost C++ libraries*. XML Press, 2014. ISBN 9781937434366 1937434362.

14. Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Professional, 4th edition, 2013. ISBN 0321563840.

15. Arthur O'Dwyer. *Mastering the C++17 STL: Make Full Use of the Standard Library Components in C++17*. Packt Publishing, 2017. ISBN 178712682X.

16. P. Deitel and H. Deitel. *C++20 for Programmers: An Objects-Natural Approach*. Deitel Developer Series. Pearson Education Canada, 2022. ISBN 9780136905691. URL https://books.google.com.br/books?id=cih_zQEACAAJ.

17. Václav Rajlich. Software evolution and maintenance. In James D. Herbsleb and Matthew B. Dwyer, editors, *Proceedings of the on Future of Software Engineering, FOSE 2014, Hyderabad, India, May 31 - June 7, 2014*, pages 133–144. ACM, 2014. . URL https://doi.org/10.1145/2593882.2593893.

18. Reno Dantas, Antonio Carvalho, Diego Marcilio, Luisa Fantin, Uriel Silva, Walter Lucas, and Rodrigo Bonifácio. Reconciling the past and the present: An empirical study on the application of source code transformations to automatically rejuvenate java programs. In Rocco Oliveto, Massimiliano Di Penta, and David C. Shepherd, editors, *25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018*, pages 497–501. IEEE Computer Society, 2018. . URL https://doi.org/10.1109/SANER.2018.8330247.

19. Walter Lucas, Rodrigo Bonifácio, Edna Dias Canedo, Diego Marcilio, and Fernanda Lima. Does the introduction of lambda expressions improve the comprehension of java programs? In Ivan do Carmo Machado, Rodrigo Souza, Rita Suzana Pitangueira Maciel, and Cláudio Sant'Anna, editors, *Proceedings of the XXXIII Brazilian Symposium on Software Engineering, SBES 2019, Salvador, Brazil, September 23-27, 2019*, pages 187–196. ACM, 2019. . URL https://doi.org/10.1145/3350768.3350791.

20. Thomas J. McCabe. A complexity measure. *IEEE Trans. Software Eng.*, 2(4):308–320, 1976. . URL https://doi.org/10.1109/TSE.1976.233837.

21. Raymond P. L. Buse and Westley Weimer. A metric for software readability. In Barbara G. Ryder and Andreas Zeller, editors, *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2008, Seattle, WA, USA, July 20-24, 2008*, pages 121–130. ACM, 2008. . URL https://doi.org/10.1145/1390630.1390647.

22. Daryl Posnett, Abram Hindle, and Premkumar T. Devanbu. A simpler model of software readability. In Arie van Deursen, Tao Xie, and Thomas Zimmermann, editors, *Proceedings of the 8th International Working Conference on Mining Software Repositories, MSR 2011 (Co-located with ICSE), Waikiki, Honolulu, HI, USA, May 21-28, 2011, Proceedings*, pages 73–82. ACM, 2011. . URL https://doi.org/10.1145/1985441.1985454.

23. Phillip Merlin Uesbeck, Andreas Stefik, Stefan Hanenberg, Jan Pedersen, and Patrick Daleiden. An empirical study on the impact of c++ lambdas and programmer experience. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, page 760–771, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450339001. . URL https://doi.org/10.1145/2884781.2884849.

24. Mingwei Zheng, Jun Yang, Ming Wen, Hengcheng Zhu, Yepang Liu, and Hai Jin. Why do developers remove lambda expressions in java? In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering*, ASE '21, page 67–78. IEEE Press, 2022. ISBN 9781665403375. . URL https://doi.org/10.1109/ASE51524.2021.9678600.

25. Lin Chen, Di Wu, Wanwangying Ma, Yuming Zhou, Baowen Xu, and Hareton Leung. How c++ templates are used for generic programming: An empirical study on 50 open source systems. *ACM Trans. Softw. Eng. Methodol.*, 29(1), jan 2020. ISSN 1049-331X. . URL https://doi.org/10.1145/3356579.

26. Filippo Ricca, Alessandro Marchetto, and Marco Torchiano. On the difficulty of computing the truck factor. In Danilo Caivano, Markku Oivo, Maria Teresa Baldassarre, and Giuseppe Visaggio, editors, *Product-Focused Software Process Improvement - 12th International Conference, PROFES 2011, Torre Canne, Italy, June 20-22, 2011. Proceedings*, volume 6759 of *Lecture Notes in Business Information Processing*, pages 337–351. Springer, 2011. . URL https://doi.org/10.1007/978-3-642-21843-9_26.

27. Amiangshu Bosu and Kazi Zakia Sultana. Diversity and inclusion in open source software (OSS) projects: Where do we stand? In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2019, Porto de Galinhas, Recife, Brazil, September 19-20, 2019*, pages 1–11. IEEE, 2019. . URL https://doi.org/10.1109/ESEM.2019.8870179.

28. Guilherme Avelino, Leonardo Teixeira Passos, André C. Hora, and Marco Túlio Valente. A novel approach for estimating truck factors. In *24th IEEE International Conference on Program Comprehension, ICPC 2016, Austin, TX, USA, May 16-17, 2016*, pages 1–10. IEEE Computer Society, 2016. . URL https://doi.org/10.1109/ICPC.2016.7503718.

29. Mívian M. Ferreira, Marco Túlio Valente, and Kecia Aline M. Ferreira. A comparison of three algorithms for computing truck factors. In Giuseppe Scanniello, David Lo, and Alexander Serebrenik, editors, *Proceedings of the 25th International Conference on Program Comprehension, ICPC 2017, Buenos Aires, Argentina, May 22-23, 2017*, pages 207–217. IEEE Computer Society, 2017. . URL https://doi.org/10.1109/ICPC.2017.35.

30. Edna Dias Canedo, Rodrigo Bonifácio, Márcio Vinicius Okimoto, Alexander Serebrenik, Gustavo Pinto, and Eduardo Monteiro. Work practices and perceptions from women core developers in OSS communities. In Maria Teresa Baldassarre, Filippo Lanubile, Marcos Kalinowski, and Federica Sarro, editors, *ESEM '20: ACM / IEEE International Symposium on Empirical Software Engineering and Measurement, Bari, Italy, October 5-7, 2020*, pages 26:1–26:11. ACM, 2020. . URL https://doi.org/10.1145/3382494.3410682.

31. Leann Myers and Maria J. Sirois. *Spearman Correlation Coefficients, Differences between.* John Wiley & Sons, Ltd, 2006. ISBN 9780471667193. . URL https://onlinelibrary.wiley.com/doi/abs/10.1002/0471667196.ess5050.pub2.

32. Joost CF De Winter, Samuel D Gosling, and Jeff Potter. Comparing the pearson and spearman correlation coefficients across distributions and sample sizes: A tutorial using simulations and empirical data. *Psychological methods*, 21(3):273, 2016.

33. Jaakko Järvi and John Freeman. Lambda functions for c++0x. In Roger L. Wainwright and Hisham Haddad, editors, *Proceedings of the 2008 ACM Symposium on Applied Computing (SAC), Fortaleza, Ceara, Brazil, March 16-20, 2008*, pages 178–183. ACM, 2008. . URL https://doi.org/10.1145/1363686.1363735.

34. Pete Becker. ISO/IEC 14882:2011 Information technology — Programming languages — C++. https://www.iso.org/standard/50372.html, 2011. [Online; accessed 03-May-2022].

35. J. Saldana. *The Coding Manual for Qualitative Researchers.* English short title catalogue Eighteenth Century collection. SAGE Publications, 2012. ISBN 9781446247372. URL https://books.google.com.br/books?id=kUms8QrE_SAC.

36. Walter Lucas Monteiro de Mendonça, José Fortes, Francisco Vitor Lopes, Diego Marcilio, Rodrigo Bonifácio, Edna Dias Canedo, Fernanda Lima, and João Saraiva. Understanding the impact of introducing lambda expressions in java programs. *J. Softw. Eng. Res. Dev.*, 8, 2020. . URL https://doi.org/10.5753/jserd.2020.744.

AUTHOR BIOGRAPHY

Walter Lucas ⊙ is a PhD student in computer science at the University of Brasília, Brazil, since 2019. He earned a master's degree in computer science from the University of Brasília, Brazil, in 2019. He obtained a degree in Information Systems at the University Center of Patos de Minas in 2015, working in Software Engineering, in the areas of Code Comprehension, Software Evolution and Program Transformations.

Rodrigo Bonifácio ⊙ is an Associate Professor at the University of Brasília, Brazil. His main research interests focus on program analysis and manipulation, with particular emphasis on software maintenance and evolution and software security.

Fausto Carvalho ● is a Software Engineer at the Brazilian Public Prosecutor's Office, with a degree in computer science from the Catholic University of Brasília, Brazil. His areas of interest are software security, programming languages, software maintenance, and evolution.

Rafael Nunes ● is a Computer Science BSc. undergraduate student at the University of Brasília. Currently, a Software Engineer working in the Communications Industry building the next generation of streaming CDNs and interested in studying Software Engineering and Programming Languages.

João Saraiva ● is an associate professor in the Department of Informatics at the University of Minho, Braga, Portugal, and a senior researcher at HASLab/ INESC TEC. He obtained a Master's degree from the University of Minho in 1993 and a PhD. Degree in Computer Science at the University of Utrecht in 1999. His areas of interest are programming language design and implementation, program analysis and transformation, and functional programming.

Paola Accioly ● holds a PhD in Computer Science from the Federal University of Pernambuco and is an Assistant Professor at the Federal University of Cariri. She obtained a Master's degree in Computer Science from the Federal University of Pernambuco (2012) and a degree in Computer Science from the Federal University of Pernambuco (2009). Her thesis focuses on Software Engineering, specifically on Collaborative Software Development Support and Empirical Software Engineering.

APPENDIX

TABLE 1 Catalog of commits that characterize rejuvenation efforts.

Project Name	Commit Date	Commit Hash	Modern feature adopted	Files modified	Additions	Deletions
akonadi	02/04/21	ddd7758e07917d7968177c0cfdaa82d620ca8d4b	Auto-typed Variables	143	709	709
akonadi	04/23/17	5535d6e5604d3bc800868ba5ea820fd6ffeb3d24	Auto-typed Variables	166	5943	13311
akonadi-contacts	11/02/20	d7bea9b44aece787d7faf42b419f6dffbd468ca1	Auto-typed Variables	78	237	237
akregator	02/03/21	b65b4e407a80324f528efa1a8bc3a6abd9082d33	Auto-typed Variables	37	73	73
akregator	11/02/20	7e6ff7d7b14c47db21b71518edbdec44367574bd	Auto-typed Variables	41	94	94
baloo-widgets	06/01/22	93b3e120c96469a28b423ddf646e397d30e3a232	Auto-typed Variables	18	55	73
gwenview	02/09/22	02ce8d7907048beb1c8aebd3d5a4b485c2c0c349	Auto-typed Variables	83	312	312
kdenlive	04/14/17	387199e1a602cd1101916ed2a96b0e8b3e8e86c5	Auto-typed Variables	178	1401	1420
kdepim-runtime	02/04/21	fdc00bcca3ad00362dc310238aae1761f6966f19	Auto-typed Variables	5	9	9
kdevelop	01/22/20	4626d69f68ea9e3af81f5324167d5608a5505c45	Auto-typed Variables	114	280	282
kid3	09/30/18	4fa1900eed4091f4100b244df07311e62208a1ab	Auto-typed Variables	65	283	557
kid3	09/29/18	f519ce07581980b95599227307eb46c8bc7f9abe	Auto-typed Variables	88	495	495
kmail	11/01/20	3055a93326cc6b90cba8c50d72b8ab0d1d33c2ac	Auto-typed Variables	130	524	524
knotes	02/04/21	0c6207800da3a16b8c0b4dda171c4cc7e4628e80	Auto-typed Variables	37	133	133
knotes	05/02/21	4d1e8e4d60e95b63286ac0103fed2e0310a3cd92	Auto-typed Variables	35	131	130
konversation	06/15/21	7c6ed0b7658a5c2ea66d6032c569012258847b91	Auto-typed Variables	63	252	252
korganizer	02/03/21	003270471b1314abf46226a92a1d32d808a83219	Auto-typed Variables	9	13	13
kpimtextedit	11/01/20	c6fa90c5c0bf719439fd9e2d2678eb0e329ff624	Auto-typed Variables	47	125	125
krfb	10/23/20	a109e3d6c9cf5e312567ec5d5d98534457ec14b9	Auto-typed Variables	7	18	19
krusader	02/24/19	59ad209a23d28ea6e745a097bf06e889b8dfb88b	Auto-typed Variables	96	455	455
okteta	02/22/19	c4c70a1e758b3fb319c0e41d3dd6ce787bf6194a	Auto-typed Variables	173	456	496
systemsettings	10/07/21	68273cc95d6760f0e4289538e19985b041bcaa5f	Auto-typed Variables	15	288	404
zanshin	02/11/15	3dfeb5fed3ceaead480444cb2be8d4e6e879ca69	Auto-typed Variables	12	18	18
calligra	03/13/21	33973e1295b	Lambda Expressions	9	33	29
kbibtex	07/20/19	40a2aee94fdc14b66b6806d552b599de3b4e64c1	Lambda Expressions	16	367	435
kwidgetsaddons	05/07/21	034e0b7b1d5af08136e38a9eb7df4ee514fe385b	Lambda Expressions	65	446	426
gwenview	07/13/21	222ed1fa0c40fb036b19bd5ecdd09a0faeb6c016	Auto-typed Variables,Lambda Expressions	6	155	138
plasma-framework	05/07/15	ea924b14691da3819ca17d876f63ffc686fcf840	Auto-typed Variables,Lambda Expressions, Range-based For	22	385	366
akonadi	12/17/16	46badf63d00f46cdb8ffcb2ff45791d4a6d63173	Range-based For	54	115	115
discover	05/21/21	13891987fbeba55d2ecc6f6ec4cf948f7c34d460	Range-based For	35	111	92
dolphin	10/23/20	a24327cd50ef17b953ecb908d260b73460158107	Range-based For	38	140	121
granatier	05/15/18	8a03ba70008f7f2c73cdd3f3d03b9ed7e922a37e	Range-based For	9	108	120
granatier	04/07/16	1535c053127e485fd6d6851f9938c677763eb00b	Range-based For	4	59	59
kalarm	08/11/19	61590fae7ba5fd8bea501c68f2f754af3f74776f	Range-based For	17	598	663
kate	09/24/19	0bbb048bd2255c1082b939c2013bf3dd0b99c2a4	Range-based For	9	15	13
kcontacts	05/10/21	3971b6278f62a9568792f7f41bd2e4ff46daf021	Range-based For	23	222	343
kdenlive	04/20/17	2e508acc340ff1d6ddda55e538c9fe01f09975f6	Range-based For	59	226	228
kdepim-runtime	12/28/16	d10456bd725e2e724c778a58a779d850f1b69420	Range-based For	29	57	58
kdesvn	07/31/19	ad2ef6dd1302af5d04a97798a6ea4b01fc60e5ee	Range-based For	1	72	76
kimap	01/20/21	c5278fe4c77329b64cef6eb097665d4b1e2e16a3	Range-based For	7	40	20
kio	10/21/19	103e13c2765e7fb587fd778c1d04c90d3af42aff	Range-based For	1	8	11
kio-extras	04/05/19	a50a8f8082baef51132562c85931c9e7ab9c7814	Range-based For	14	45	64
kleopatra	02/27/17	96409339	Range-based For	9	19	16
konsole	11/24/19	c83bb19a68a65a59e149586d809c442168ba35aa	Range-based For	13	55	51
konversation	10/19/20	9f9bf118	Range-based For	6	27	43
kphotoalbum	04/10/20	233c3d7fc43217f6acf2a4e2db915bf5747cb50c	Range-based For	53	206	206
ktorrent	06/29/17	7f8795555185226bb17909b8a954592fa7c947de	Range-based For	14	66	66
ktorrent	07/06/17	2ad882b61101dfe8414e8f47f89360f4e2d96754	Range-based For	10	47	45
kwin	08/08/19	91faa589c70f9ca9a9d518ac072e186846c88827	Range-based For	1	184	112
labplot	12/03/17	5c28a3e5f994fbc586519fad3ec2aadff7357070	Range-based For	36	388	389
marble	01/27/17	33eff9bc4ea9581dc3d84783f62747463f2c047e	Range-based For	176	537	532
okular	03/26/19	f34ebf659f6cd93233cca345d873bd54c039b83a	Range-based For	3	50	90
partitionmanager	08/31/16	e7ac5e5fa2800c9583ec1223abd1fce2a6e51a50	Range-based For	6	30	16
krusader	02/24/19	15ca9d93c97b0729e29c74f38e6cdeb31793d09c	Auto-typed Variables, Range-based For	22	148	153
amarok	09/26/20	ce93fb34c396f1b7766daaf43669d82f949ff091	Auto-typed Variables	104	7170	783
kdevelop	10/25/18	c50f4442ccbb6ecb95d2e0d7f0484060bddb5747	Lambda Expressions	343	1224	1224
kwin	07/14/18	049c6e0966ceda533e023a22f29a5d43a65cef0c	Auto-typed Variables	1	4	4