2023 IEEE International Conference on Software Maintenance and Evolution (ICSME) | 979-8-3503-2783-0/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICSME58846.2023.00082

Understanding the Motivations, Challenges, and Practices of Software Rejuvenation

Walter Lucas
Computer Science Department
University of Brasília
Brasília, Brazil
walter.mendonca@aluno.unb.br

Rodrigo Bonifácio
Computer Science Department
University of Brasília
Brasília, Brazil
rbonifacio@unb.br

João Saraiva
Department of Informatics
Universidade do Minho
Braga, Portugal
saraiva@di.uminho.pt

***Abstract*—The continuous evolution of programming languages has brought benefits and new challenges for software developers. In recent years, we have witnessed a rapid release of new versions of mainstream programming languages like Java. While these advancements promise better security, enhanced performance, and increased developers' productivity, the constant release of new language versions has posed a particular challenge for practitioners: how to keep their systems up-to-date with new language releases. This thesis aims to understand the pains, motivations, and practices developers follow during rejuvenating efforts—a particular kind of software maintenance whose goal is to avoid obsolesce due to the evolution of programming languages. To this end, we are building and validating a theory using a mixed methods study. In the first study, we interviewed 23 software developers and used the Constructivist Grounded Theory Method to identify recurrent challenges and practices used in rejuvenation efforts. In the second study, we mined the software repositories of open-source projects written in C++ and JavaScript to identify the adoption of new language features and whether or not software developers conduct large rejuvenation efforts. The first study highlights the benefits of new feature adoption and rejuvenation, revealing developer methods and challenges. The second study emphasizes open-source adoption trends and patterns for modern features. In the third and final study, our goal is to share our theory on software rejuvenation with practitioners through the Focus Group method with industrial patterns.**

***Index Terms*—Grounded Theory, Software Rejuvenation, Software Engineering, Software Evolution**

I. Introduction

Software systems are in a state of constant evolution, driven by different reasons, including not only domain and technology evolution, but also the recent short release cycles of new versions of mainstream programming languages. As software languages progress, they bring forth a plethora of new features and enhancements, promising improved security, performance, and developer productivity [1]–[3]. To illustrate the short release cycle of programming languages, from 2014 to 2023, twelve new Java language versions were officially released (Java 8 — Java 20). Contrasting, from 1996 to 2014, only eight major releases of the Java language have been made available (Java 1.0 to Java 8). However, with this rapid evolution comes a set of challenges that developers must deal with, particularly when it comes to software maintenance and the intricacies of legacy systems [4]. In particular, it is hard to maintain a system that contains new and old program language constructs and idioms that implement the same concern. Failure to rejuvenate a system might lead the software to age, resulting in obsolescence, rendering once-vibrant applications ineffective in the face of modern demands [4].

Therefore, software systems can benefit from source code rejuvenation, a process that aims at transforming the source code to support more recent features of a programming language [5]. The rejuvenation of legacy code is vital to avoid obsolescence and maintain compatibility with modern software. Recent research [6], [7] presented a set of benefits related to software rejuvenation (including increased flexibility, agility, and productivity). Lucas et al. [2], [8] suggested that replacing legacy constructs (such as anonymous inner classes and `foreach` statements) with lambda expressions improves the understanding of Java programs in specific scenarios and can make the code more concise.

Despite its potential benefits, the motivations and challenges developers face are shortly unexplored in the existing literature, creating a significant gap in understanding the strategies to overcome the side effects of software aging. In particular, as initial results of our research suggest, the demand to address critical updates and bug fixes often takes precedence over software rejuvenation, postponing these efforts for years in some cases. This Ph.D. thesis explores the following research questions to address this literature gap.

RQ1: **What are the motivations that lead software developers to rejuvenate their software?** This question aims to explain the factors, phenomena, and situations that motivate software developers to rejuvenate their code;

RQ2: **What are the challenges that hinder software developers from rejuvenating their software?** This question aims to explain which situations and factors make it difficult or prevent developers to rejuvenate their programs.

II. Thesis Outline

This thesis aims to build a theory on the pains, motivations, and practices developers follow during rejuvenating efforts—a particular kind of software maintenance whose goal is to avoid software obsolesce due to the evolution of programming languages. To achieve this goal, we have conducted two studies so far. In the first, we interviewed 23 software practitioners and, using the Constructivity Ground Theory Method, we identified recurrent challenges and practices used in rejuvenation efforts. We detail the first study in Section IV. This research is in the final stage, lacking some revisions of the results and the conclusion of writing a scientific paper (to be submitted by the end of September 2023).

In the second study our goal was to mine the source code repository of C++ and JavaScript open source projects to find evidence that developers replace legacy programming language constructs and idioms by new ones and even conduct large rejuvenation efforts. Regarding the C++ study, we conducted a large-scale exploratory study of 272 C++ projects from the

2576-3148/23/$31.00 ©2023 IEEE
DOI 10.1109/ICSME58846.2023.00082

KDE community [9]. The KDE Open Community is a globally recognized open-source community that is actively engaged in the development of a wide range of applications across various domains, such as games, education, and development frameworks. We analyzed the adoption of a set of modern C++ features and then contacted developers participating in code rejuvenation efforts we mined. Our findings have multiple ramifications. First, we present a list of benefits and rejuvenation scenarios that C++ programmers can implement, indicating that using *lambda expressions*, *range-based for loops*, and *auto-typed variables* improves code readability and software maintenance.

In addition, we report that automated tools such as Clazy, Clang-tidy, KDevelop, and Clion assist KDE developers in identifying opportunities for rejuvenation. Finally, the set of contributions that characterize rejuvenation efforts provides valuable examples for both developers and tool builders. Improving software development practices, the C++ study provides insights and recommendations that can benefit both the KDE community and a broader audience of C++ developers. We replicated the C++ study to characterize the trends in the adoption of modern JavaScript features. As such, we mined the source code history of 100 JavaScript open-source projects. This study offers insights into the adoption patterns and trends for modern JavaScript features within open-source communities' applications. Our initial findings underscore the widespread incorporation of these features, underscoring their pivotal role in present-day development practices. The early integration of specific features revealed that JavaScript developers have a forward-looking mindset and eagerness to infuse quality improvements into their projects, thereby rejuvenating their codebases. Although we have already published the results of the C++ study [9], our plan is to submit the results of the JavaScript study in October 2023. We detail the second study in Section V.

These studies aim to improve and validate our theory about software rejuvenation efforts through new data found in the history of the analyzed software repositories, which bring evidence to support the facts presented in our theory. We are planning a third and final study of this research to collect evidence from the industry about the usefulness of our theory. We intend to apply the focus group method [10] to evaluate the emerging theory and refine it according to the new data collected. The objective is that the MSR studies on C++ and JavaScript and the focus group study will help us improve and validate our theory. During the next twelve months (from August 2023 to July 2024), I intend to conduct the third study and then (a) consolidate the results of our research, (b) finish writing my thesis, and (c) present my research to the defense committee.

Research Method. As we mentioned in this section, this research involves three studies that use different methods.

- **First Study** aims to build a theory on software rejuvenation using the Constructivity Grounded Theory method [11], [12].
- **Second Study** aims to identify (a) how C++ and JavaScript developers replaces legacy constructs by modern language features and (b) whether or not they conduct large rejuvenation efforts. The second study relies on mining software repository efforts [1], [9].
- **Third Study** aims to improve and validate our theory on software rejuvenation efforts with the feedback from industry. To this end, our goal is to use the Focus Group method [10] that should be conducted in at least three companies.

III. Background and Related Works

Software maintenance can be comprehended as a collection of discrete alterations carried out on a software system during its entire life cycle [13]. According to the ISO/IEC/IEEE 14764:2022 [14] standard, a basic process for software maintenance may contain the following phases: process implementation, problem analysis, and modification, modification implementation, revision /acceptance of software maintenance, migration, and retirement.

Bragagnolo et al. [7] discusses the concepts of modernization, such as legacy systems, their decline, approaches to recovering from decline, and the material relation between these processes and software engineering. The authors present a taxonomy of software migrations that can be performed in a program that include, but not limited to *Web Migration*, which involves the transition from a desktop application (client-server) to a web application [15], *Service Migration* when application functionality can be made available as a service [16], *Library Migration*, which consists of delegating a concern to a given library/structure [17], *Language migration*, which involves translating source code from one programming language to another (e.g. Java to Kotlin [18]), and *Paradigm migration*, which implies changes in the organization and semantics of the code (e.g. from the procedural to object-oriented paradigm [19]). The study results in a theory that unifies the acknowledgment of methods, processes, and planning from legacy systems migrations.

Khadka et al [6] present a set of benefits related to software migration/modernization. They conducted five retrospective case studies of software modernization were analyzed in an empirical study. The researchers assessed whether the business objectives established before the modernization activities were met. According to the study, the goals were only partially met, and there are benefits (Like Increased flexibility, Increased agility, and Cost reduction) and negative consequences (decreasing of performance and user resistance) resulting from modernization efforts. These studies provide a classification system for various types of migrations, as well as a compilation of advantages and disadvantages associated with them. However, they do not delve into the subject of source code rejuvenation driven by the evolution of programming languages.

In addition, Pirkelbauer et al [5] present the migration between versions of the same programming language called source code rejuvenation, which consists of evolving the source code of the programs through code transformations to support the modern features of programming languages. In this thesis, we are interested exclusively in this kind of migration.

Also, studies in the literature report that developers can benefit from tools of automated code transformations to replace legacy constructions with modern programming language features [20]–[22]. Kumar et al [20] present a tool set to rejuvenate C++ libraries by replacing macros with modern constructs introduced into the C++11 standard library. Dantas et al [21] present a library of Java transformations developed in the RASCAL language [23] that rejuvenates legacy systems to support Java programming language constructs in version 8. The study revealed that simple transformations such as introducing the diamond operator are more likely to be accepted than substantial transformations that substantially change the code, such as enhanced refactor loops

for the new functional style. Mazinanian et al. [1] reported the need for improvements in Java language design and suggested that developers make changes manually and need smarter tool support with suggestions for appropriate code changes.

The previous studies highlight the advantages and difficulties associated with the use of automated tools for code rejuvenation. However, these studies do not address the specific challenges that developers encounter when engaging in rejuvenation activities.

IV. A Theory about Software Rejuvenation

In our study, we seek to better understand how industry professionals evolve their software from the evolution of programming languages. We use the Grounded Theory approach (GT) [11] to find factors that can block, delay, and motivate the evolution of software to adopt modern features of new versions of programming languages. We use GT because it allows us to better understand social interactions and behaviors, as software is mostly developed and maintained by humans. GT is also suitable in underexplored areas and research on software evolution motivated by language evolution in the view of professionals is still a mystery to software engineering. Among the approaches existent, we will follow the steps proposed by [11], as we believe that to conduct a robust and in-depth investigation in software engineering, it is necessary to have knowledge and skills beyond those normally available to social science researchers. Contrary to the previous strands, the constructivist GT considers the experience and knowledge of researchers in the explored area.

A. Study Settings

In this study, we present a comprehensive grounded theory study aimed at understanding the motivations and challenges behind code rejuvenation among professional developers. Our data collection process involved conducting semi-structured interviews with experienced developers, focusing on their experiences with software and programming language evolution. To ensure the validity of our interview script, we conducted a pilot study with five developers to validate the open-ended questions. Our target population consisted of experienced developers with at least four years of industry experience, as they are more likely to encounter software and programming language evolution throughout their careers.

Initially, we sought to recruit developers with three or more years of experience contributing to popular open-source projects on GitHub. However, this strategy did not yield the desired results. Consequently, we adopted alternative methods, such as reaching out to developers through social networks like LinkedIn and sending emails to individuals in our contact network. We conducted interviews with a total of 23 developers, employing theoretical sampling [11] to ensure adequate data representation.

The demographic data of the participants, including their identification (**Id**), years of experience (**Exp**), role at work (**Role**), and domain of the participant company (**Domain**), are presented in Table I. The next sections refer to specific participants using the assigned IDs. Next, all 23 interviews were transcribed (a total of 82100 words).

The initial coding phase involved analyzing the interview data, resulting in 80 unique codes and 441 codifications. We derived codes such as ***improving code maintenance*** and ***improving code comprehension*** from a developer's reported benefits of rejuvenating code. Memoing, an essential aspect of the coding and analysis process, involved taking informal notes about participants, phenomena, and the investigation. We crafted a memo reflecting the motivation for code rejuvenation and associated the codes ***improving code maintenance*** and ***improving code comprehension*** with the category ***quality improvements***.

To ensure consistency and accuracy, we conducted constant comparisons throughout the analysis process, involving two researchers. The focused coding phase involved a systematic evaluation of the codes obtained during the initial coding phase to identify categories. Based on participants' responses, we observed that rejuvenating source code could lead to various benefits. Consequently, we grouped related codes under the category ***quality improvements***, which encompassed six unique codes: ***improving code maintenance***, ***improving code comprehension***, ***improving software architecture***, ***improving code performance***, ***new language features might improve correctness***, and ***new language features can improve security***.

Theoretical sampling played a role in shaping our interview questions, enabling us to delve deeper into participants' significant experiences, practices, obstacles, and perspectives. We included additional questions in subsequent interviews to explore specific aspects further. Theoretical saturation signified the end of data collection when the coding process, refinement, and re-interviews no longer yielded new insights or theoretical knowledge. With this, we finalized our coding process, resulting in 63 unique codes and 438 codifications, thereby attaining a coherent grounded theory consistent with the data.

TABLE I
Grounded theory participant demographics.

Id	Exp	Role	Domain
P1	12 years	Senior Developer	Public Prosecutor's Office
P2	17 years	Software Architect	Consultancy
P3	18 years	Software Architect	Enterprise software
P4	6 years	Developer	Software tooling
P5	5 years	Developer	Enterprise software
P6	12 years	Senior Developer	Public Prosecutor's Office
P7	10 years	Tech lead	Enterprise software
P8	6 years	Developer	Military Institute of Engineering
P9	5 years	Developer Consultant	Software tooling
P10	6 years	Developer	Enterprise software
P11	17 years	Senior Developer	Enterprise software
P12	years	Senior Developer Researcher	Institute of Research Software tooling
P13	4 years	Developer	Military Institute of Engineering
P14	20 years	Senior Developer	Enterprise software
P15	11 years	Tech lead Co-Founder	Enterprise software
P16	30 years	Founder	Enterprise software
P17	9 years	Tech lead	Enterprise software
P18	5 years	Developer	Public Prosecutor's Office
P19	8 years	Senior Developer	Public Prosecutor's Office
P20	25 years	Senior Developer	Enterprise software
P21	15 years	Senior Developer	Banking Health
P22	30 years	Professor Senior Developer	Enterprise software Academic
P23	6 years	Researcher Developer	Enterprise software Academic

B. Results

In this section, we introduce the codes of theoretical categories that surfaced throughout our investigation. We structure this section based on the *motivations*, *challenges*, and *practices* developers adopt during code rejuvenation endeavors. We emphasize the categories and codes, including direct quotes from the interviews.

Also, we present the number of occurrences for codes in the text Superscript ([*]) with color blue.

1) Motivations

The study participants emphasize that code rejuvenation efforts can be instrumental in ***preventing software obsolescence***[8] (P1, P3, P6, P11, and P19). Obsolescence becomes a concern when support for a specific version of a programming language is discontinued, and programs continue to utilize that version. In such cases, developers feel compelled to rejuvenate the software. An essential motivation for rejuvenating software is to enhance the overall program quality. Participants noted that new language features often improve both **security**[9] (P3, P12, P17, 21, and P22) and **performance**[12] (P2, P3, P8, P11, P12, P17, 22, and P23) of systems. Minor language revisions, such as Java SE 13.0.1 and 13.0.2, often include security patches and bug fixes for standard libraries. Furthermore, the Java Platform Module System [24], a prominent feature in Java 9, aims to enhance the scalability of the Java SE Platform and boost performance by employing whole-program optimization techniques for complete configurations of platform, library, and application components [24].

P2

> "*Yes, it is important. Mainly because of security issues, performance issues, and code readability issues.)*"

Participants emphasized that code rejuvenation efforts can significantly improve the internal quality attributes of a system, particularly through reductions in **boilerplate code**[6] (P1, P7, P12, P14, and P19) and advancements in **program comprehension**[34] (P1-P7, P10-P14, P16-P20, P22, and P23). The introduction of new language constructs aimed at reducing verbosity allows developers to achieve the same functionality with fewer lines of code, streamlining the codebase and facilitating software maintenance and management. Rejuvenation efforts also have positive effects on **developer productivity**[16] (P3, P4, P8, P10, P11, P13, P14, P16, P17, P19 and P20) and **software correctness**[13] (P3, P8, P11, P16, P17, P19, P21 and P22). For instance, the JDK Enhancement Proposal 359 (JEP395) introduces the `record` construct as an alternative to certain Lombok features, allowing developers to migrate legacy code and reduce library dependencies, further enhancing productivity.

2) Challenges

Despite the numerous benefits of rejuvenation, developers encounter challenges that may impede their efforts. Time constraints arise from the need to **modern language features need some time for being adopted**[28] (P2, P3, P9, P10, P12-P15, P18-P20, P12, and P23), and **It is hard to keep updated with language evolution**[14] (P1, P3, P6, P7, P10, P15, P16, P18, P20, P21 and P23), which require developers to allocate time for understanding and incorporating these features effectively. The findings of our mining study [9] indicate that the prompt and extensive integration of modern language features is not instantaneous. The findings of our study indicate that the widespread adoption of modern features occurred approximately five years after the release of the C++11 specification.

Keeping pace with the rapid evolution of programming languages, like C++ and Java, also presents challenges, particularly when dealing with multiple languages for different system components. For example, a new version of C++ is released every three years; while a new version of the Java language is being released every 6 months [1]. Another key challenge is ***balancing development and maintenance time***[16] (P4, P6, P8, P9, P11, P13-P15, P18, and P20-P22), where developers must strike a delicate equilibrium between implementing new features and maintaining existing code. Additionally, cultural aspects within organizations may undervalue maintenance tasks like rejuvenation, hampering the prioritization of these efforts.

P21

> "*The main issue we had there was time, as I worked alone and had to meet many demands at the same time. And a process like this, of renewing a code construction that's not very good is not that fast.*"

Furthermore, developers perceive certain factors that may *prevent software rejuvenation*. The difficulty of ***migrating large legacy codebases***[16] (P1, P4, P5, P7, P8, P11, P13-P15, P17, and P20-P22) to the latest language version stands as a significant obstacle, particularly due to the complexity involved in such migrations. Additionally, the apprehension of introducing ***bugs during rejuvenation***[22] (P2, P3, P8, P9, P11-13, P16-P18, P17, and P20-P22) creates reluctance among developers to embark on rejuvenation efforts. However, developers are aware of incremental migration strategies to minimize these challenges.

3) Practices

Developers adopt diverse approaches to rejuvenating software, with differing opinions on the ideal timing for rejuvenation. Some ***prioritize software stability***[4] (P1, P10, P20, and P22) over staying up-to-date with language versions, while others advocate for prompt rejuvenation to reduce migration efforts and associated risks. Factors like ***implementing new features***[4] (P4, P9, and P18), altering system architecture, encountering critical bugs, or facing the discontinuity of language support influence the decision to rejuvenate. A reactive-based approach may emerge when reconciling rejuvenation with ongoing development proves challenging. However, practices are influenced by organizational and product-specific considerations, and ***developers must carefully weigh the benefits and risks of rejuvenation efforts***[9] (P12, P13, P15, P16, P18, P22 and P23).

P12

> "*In general, the best attitude from my point of view is: to what extent is it worth changing a project in order to evolve a language to reach a new feature or new functionality? The impact on the project must be taken into consideration.*"

Most participants in the study rejuvenate their systems without specialized tools, despite the availability of IDEs (such as IntelliJ and Eclipse) and command-line tools for support. Reasons for this vary; some developers rely on ***manual rejuvenation***[15] (P1-P3, P11-P13, P15, P16, and P20-P23) with confidence in their

[1]https://blogs.oracle.com/javamagazine/post/java-long-term-support-lts

test suites, while others only ***use static analysis tools to identify opportunities for rejuvenation***[15] (P3, P4, P10, P11, P14, P15, P20, P22 and P23). The lack of awareness about helpful rejuvenation tools and concerns about potential risks and the need for tool configuration contribute to the preference for manual efforts. Overall, developers' approaches to rejuvenation and tool adoption are shaped by their confidence in manual efforts, familiarity with available tools, and risk tolerance for automated code changes.

V. The Adoption of New Language Features

In the Second study, we conducted an analysis on the software repositories of open-source projects implemented in C++ and JavaScript in order to ascertain the extent to which new language features are adopted and whether software developers realize rejuvenation endeavors. These studies are guided by some questions:

(Q1) To what extent do software systems rely on modern features?
(Q2) When did software developers start using modern features?
(Q3) Is there any trend in the adoption of modern features in open-source applications?
(Q4) Do software developers conduct maintenance efforts having the sole goal of rejuvenating their code?
(Q5) Which tools do software developers use to support maintenance efforts for code rejuvenation?
(Q6) What are the reasons that motivate software developers to conduct maintenance efforts for code rejuvenation?
(Q7) Are the Core Developers of the projects responsible for conducting rejuvenation efforts?

Exploring these research questions enhances our grasp of modern C++ and JavaScript feature adoption, temporal transitions in project codebases, and potential code rejuvenation trends. It clarifies developer involvement, what tools for rejuvenation are used or not, motivations that lead developers to rejuvenate their programas, and identifies key contributors.

A. Study Settings

We analyzed C++ repositories that we checked out from the GitHub of The KDE Open Community [9]. 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 dataset contains 272 KDE programs and libraries written in C++. We analyzed the adoption of a set of modern features such as *lambda expressions*, *auto-typed variables*, *range-based for*, and contacting developers participating in code rejuvenation efforts.

Next, we replicate the study to characterize the trends in the adoption of modern JavaScript features. The dataset containing the 100th most rated JavaScript repositories was selected using the SEART tool [25]. The parameters of the search were JavaScript projects whose number of commits was more than 1000 (one thousand), the creation date to be before 2012 with at least one update on January of 2023. We analyzed the adoption of a set of modern features such as *Async Declarations*, *Const Declarations*, *Arrow Function Declarations*, *Let Declarations*, and *Object Destructuring*.

B. Results

Regarding the C++ study, Our findings suggest that KDE developers extensively use *auto-typed variables*, *lambda expressions*, and *range-based for* in more than 60% of the projects [9]. Futhermore, the findings of our study indicate that the integration of new language features does not occur instantaneously, as it took approximately five years for the widespread adoption of modern features following the release of C++11. The results of our study also indicate a consistent pattern of growing acceptance and adoption of modern C++ language features within KDE projects. Additionally, there is a noticeable inclination towards transitioning existing codebases to incorporate *auto-typed variables*, *lambda expressions*, and *range-based for*.

KDE developers actively pursue rejuvenation efforts for code modernization, notably adopting new features through refactoring. Our study revealed over 50 such endeavors, impacting hundreds of lines in C++ code. Our findings also indicate that KDE developers employ tools (Clazy, Clang-tidy, KDevelop, Clion) for rejuvenation, mainly identifying scenarios for improvement. Automation aids significant efforts, reducing costs, particularly for *range-based for* and *auto-typed variables*; however, *lambda expressions* incorporation demands extra cognitive effort, per developer reports.

Finally, KDE developers leverage modern C++ features (*lambda expressions*, *range-based for*, *auto-typed variables*) for enhanced code readability and reduced errors, streamlining maintenance. Code rejuvenation serves dual purpose: attracting contributors and extending software lifespan. Applying the Truck Factor method [26], we observed 63.2% (36 out of 57) major rejuvenation commits linked to core project developers. This underscores the significance of technical leaders' engagement in extensive maintenance for effective software rejuvenation.

In relation to the JavaScript study, our results suggest that, JavaScript developers extensively employ *Arrow Function Declarations*, *Const Declarations*, *Async Declarations*, *Let Declarations*, and *Object Destructuring* (adoption rates: 97%, 91%, 88%, 86%, 75%). Additionally, *Default Parameters*, *Await Declarations*, *Promise Declarations*, *Import Statements*, and *Spread Arguments* are significantly used (45% to 60%), while *Class Declarations*, *Rest Statements*, *Export Declarations*, and *Promise All() and Then()* exhibit moderate adoption (33% to 50%). These findings illuminate JavaScript feature usage patterns, emphasizing key features in modern development. JavaScript developers quickly embraced *Arrow Function Declarations* and *Async Declarations*, signifying early adoption of these modern features. Additionally, other modern JavaScript features gained widespread acceptance within 1-2 years of their introduction in new JavaScript versions.

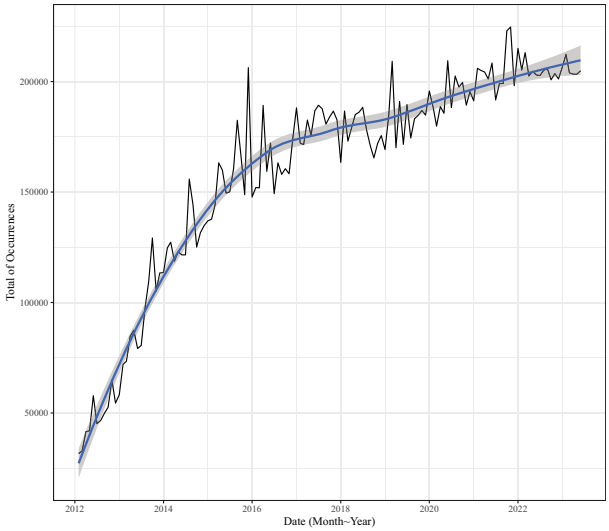
Finally, Our analysis of JavaScript apps reveals a notable rising adoption trend for various modern JavaScript features, confirmed by significant upward trends in regression analysis. For example, depicted in Figure 1, underscore the consistent and growing utilization of *Arrow Function Declarations* over time.

Our plan is to finish the analysis and consolidate all results to submit the study in October 2023.

VI. Final Remarks

The outcomes of this thesis enrich our comprehension of the ever-evolving realm of software development, underscoring remaining updated on novel features to harness their potential optimally. Our study provides insights into code rejuvenation's

Fig. 1. Distribution of *Arrow Function Declarations* usage across the JavaScript projects in our dataset.



motivations and challenges. It highlights benefits and hurdles like time constraints, evolving features, and organizational culture, aiding developers in navigating the process for enhanced software practices. With Programming language's ongoing evolution, forthcoming studies can leverage these findings to delve into the effects of these features on software caliber, developer efficiency, and overall application performance. In our future works , we aims to investigate the efforts made by developers in incorporating the most recent language features. This will result in the creation of a comprehensive collection of recommended practices and transformations. The purpose of this catalog is to assist fellow developers, communities, and organizations in maintaining the integrity of their existing programs and enhancing their practicality.

REFERENCES

[1] D. Mazinanian, A. Ketkar, N. Tsantalis, and D. Dig, "Understanding the use of lambda expressions in java," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, pp. 1–31, 2017.

[2] W. Lucas, R. Bonifácio, E. D. Canedo, D. Marcilio, and F. Lima, "Does the introduction of lambda expressions improve the comprehension of java programs?" in *Proceedings of the XXXIII Brazilian Symposium on Software Engineering, SBES 2019, Salvador, Brazil, September 23-27, 2019*. ACM, 2019, pp. 187–196. [Online]. Available: https://doi.org/10.1145/3350768.3350791

[3] J. L. Overbey and R. E. Johnson, "Regrowing a language: refactoring tools allow programming languages to evolve," in *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, 2009, pp. 493–502.

[4] D. L. Parnas, "Software aging," in *Proceedings of 16th International Conference on Software Engineering*. IEEE, 1994, pp. 279–287.

[5] P. Pirkelbauer, D. Dechev, and B. Stroustrup, "Source code rejuvenation is not refactoring," in *International Conference on Current Trends in Theory and Practice of Computer Science*. Springer, 2010, pp. 639–650.

[6] R. Khadka, P. Shrestha, B. Klein *et al.*, "Does software modernization deliver what it aimed for? A post modernization analysis of five software modernization case studies," in *2015 IEEE International Conference on Software Maintenance and Evolution, ICSME 2015, Bremen, Germany, September 29 - October 1, 2015*. IEEE Computer Society, 2015, pp. 477–486. [Online]. Available: https://doi.org/10.1109/ICSM.2015.7332499

[7] S. Bragagnolo, N. Anquetil, S. Ducasse, A. Seriai, and M. Derras, "Software migration: A theoretical framework (a grounded theory approach on systematic literature review)," *Empirical Software Engineering*, 2021.

[8] W. L. M. de Mendonça, J. Fortes, F. V. Lopes *et al.*, "Understanding the impact of introducing lambda expressions in java programs," *J. Softw. Eng. Res. Dev.*, vol. 8, 2020. [Online]. Available: https://doi.org/10.5753/jserd.2020.744

[9] W. Lucas, F. Carvalho, R. C. Nunes *et al.*, "Embracing modern c++ features: An empirical assessment on the kde community," *Journal of Software: Evolution and Process*, vol. n/a, no. n/a, p. e2605, 2023. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.2605

[10] J. Kontio, J. Bragge, and L. Lehtola, "The focus group method as an empirical tool in software engineering," in *Guide to Advanced Empirical Software Engineering*, F. Shull, J. Singer, and D. I. K. Sjøberg, Eds. Springer, 2008, pp. 93–116. [Online]. Available: https://doi.org/10.1007/978-1-84800-044-5_4

[11] K. Charmaz, *Constructing grounded theory : a practical guide through qualitative analysis*. London; Thousand Oaks, Calif.: Sage Publications, 2006. [Online]. Available: http://www.amazon.com/Constructing-Grounded-Theory-Qualitative-Introducing/dp/0761973532

[12] ——, "Grounded theory as an emergent method," in *Handbook of emergent methods*, S. N. Hesse-Biber and P. Leavy, Eds. New York, NY: Guilford Press, 2008, pp. 155–170.

[13] H. C. Benestad, B. Anda, and E. Arisholm, "Understanding software maintenance and evolution by analyzing individual changes: a literature review," *J. Softw. Maintenance Res. Pract.*, vol. 21, no. 6, pp. 349–378, 2009. [Online]. Available: https://doi.org/10.1002/smr.412

[14] ISO/IEC/IEEE, "Iso/iec/ieee international standard - software engineering - software life cycle processes - maintenance," *ISO/IEC/IEEE 14764:2022(E)*, pp. 1–46, 2022.

[15] A. De Lucia, R. Francese, G. Scanniello, and G. Tortora, "Developing legacy system migration methods and tools for technology transfer," *Software: Practice and Experience*, vol. 38, no. 13, pp. 1333–1364, 2008. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.870

[16] M. Razavian and P. Lago, "A lean and mean strategy for migration to services," in *Proceedings of the WICSA/ECSA 2012 Companion Volume*, ser. WICSA/ECSA '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 61–68. [Online]. Available: https://doi.org/10.1145/2361999.2362009

[17] H. Zhou, J. Kang, F. Chen, and H. Yang, "Optima: An ontology-based platform-specific software migration approach," in *Seventh International Conference on Quality Software (QSIC 2007)*, 2007, pp. 143–152.

[18] M. Martinez and B. G. Mateus, "Why did developers migrate android applications from java to kotlin?" *IEEE Trans. Software Eng.*, vol. 48, no. 11, pp. 4521–4534, 2022. [Online]. Available: https://doi.org/10.1109/TSE.2021.3120367

[19] Y. Zou and K. Kontogiannis, "A framework for migrating procedural code to object-oriented platforms," in *Proceedings Eighth Asia-Pacific Software Engineering Conference*, 2001, pp. 390–399.

[20] A. Kumar, A. Sutton, and B. Stroustrup, "Rejuvenating C++ programs through demacrofication," in *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 2012, pp. 98–107.

[21] R. Dantas, A. Carvalho, D. Marcilio *et al.*, "Reconciling the past and the present: An empirical study on the application of source code transformations to automatically rejuvenate java programs," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2018, pp. 497–501.

[22] L. Franklin, A. Gyori, J. Lahoda, and D. Dig, "LAMBDAFICATOR: from imperative to functional programming through automated refactoring," in *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, D. Notkin, B. H. C. Cheng, and K. Pohl, Eds. IEEE Computer Society, 2013, pp. 1287–1290. [Online]. Available: https://doi.org/10.1109/ICSE.2013.6606699

[23] P. Klint, T. van der Storm, and J. J. Vinju, "RASCAL: A domain specific language for source code analysis and manipulation," in *Ninth IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2009, Edmonton, Alberta, Canada, September 20-21, 2009*. IEEE Computer Society, 2009, pp. 168–177. [Online]. Available: https://doi.org/10.1109/SCAM.2009.28

[24] JSR-376 Expert Group, "Java platform module system (jsr 376)," OpenJDK, Tech. Rep., 2017. [Online]. Available: https://openjdk.org/projects/jigsaw/spec/

[25] O. Dabic, E. Aghajani, and G. Bavota, "Sampling projects in github for MSR studies," in *18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021*. IEEE, 2021, pp. 560–564.

[26] G. Avelino, L. Passos, A. Hora, and M. T. Valente, "A novel approach for estimating truck factors," in *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*. IEEE, 2016, pp. 1–10.