Institute of Software Engineering
University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Exploring the Adoption of Java Version Features and Their Relationship to Software Quality on GitHub

Daniel Abajirov

Course of Study: Softwaretechnik

Examiner: Prof. Dr. Stefan Wagner

Supervisor: Dr. Justus Bogner, Ms. Umm-e-Habiba

Commenced: May 2, 2023

Completed: November 2, 2023

Abstract

The progression of software development has driven the continuous evolution of programming languages, of which Java is an example with its many major versions since 1996. With each new version, performance, usability, and robustness are improved to meet the challenges of modern software development. However, while there is an understanding of the enhancements each version brings, there is a lack of modern research on the practical benefits and impacts of these features on software quality, as well as the adoption rate of these version features.

Our research utilized a combination of extensive data mining techniques and a quantitative survey approach. We analyzed 2,167 GitHub repositories spanning three major Java LTS versions (8, 11, 17). To supplement our findings, we conducted a survey involving 47 developers with varying years of experience, utilizing close-ended questions to derive quantitative insights that could be compared with our primary research results.

We observed a surprising underutilization of Java built-in methods. On average, only 1.03 methods were used per KLOC. Interestingly, lower usage was observed in versions 8 to 17, while Java 6 showed significant method adoption. These results are in contrast to the survey results where developers reported high usage of Java built-in methods.
Furthermore, we found that the impact on software quality, as measured by the number of bugs and code smells per KLOC, was minimal. Domain selection also does not appear to have a significant impact on the use of Java built-in methods.

The divergence between the practices in open-source projects on GitHub and developers' perceptions indicates a layered complexity in software development practices. Our study underscores a need for increased awareness and targeted strategies to fully harness the evolving capabilities of Java versions.

Contents

1 Introduction **13**
- 1.1 Motivation . . . 13
- 1.2 Research Questions . . . 14
- 1.3 Structure of the Thesis . . . 15

2 Background **17**
- 2.1 Java . . . 17
- 2.2 Software Quality . . . 18

3 Related Work **21**

4 Methodology **25**
- 4.1 Study Object and Sampling . . . 25
- 4.2 Data Collection . . . 27
- 4.3 SonarQube Analysis . . . 31
- 4.4 Identifying Project Domains . . . 33
- 4.5 Mapping Process . . . 35
- 4.6 Survey . . . 36
- 4.7 Metrics . . . 40

5 Results **41**
- 5.1 Adoption of Java Version Features (RQ1) . . . 41
- 5.2 Java Feature Adoption and Software Quality (RQ2) . . . 47
- 5.3 Factors Influencing Java Feature Adoption: Domain, Contributors, and Project Size (RQ3) . . . 49
- 5.4 Java Developers' Experiences vs. Observed Adoption Trends (RQ4) . . . 55

6 Discussion **63**

7 Threats to Validity **67**
- 7.1 Internal Validity . . . 67
- 7.2 External Validity . . . 69
- 7.3 Construction Validity . . . 69
- 7.4 Conclusion Validity . . . 70

8 Conclusion **73**

Bibliography **75**

List of Figures

4.1 Process Overview: Collecting Java Methods Software Quality Data 32
4.2 Process Overview: Collecting Data on Java Built-in Methods Usage 36

5.1 Distribution of Java Versions Weighted by Method Occurences 42
5.2 Distribution of Java Built-in Methods Usage Across Analyzed Projects 43
5.3 Distribution of the Java Built-in Methods per Version over the years 47
5.4 Distribution of the Java Built-in Methods for Version 8 over the years 48
5.5 Distribution of the Java Built-in Methods per Version 11 and 17 over the years . . 49
5.6 Distribution of Code Smells and Bugs for the individual Java Versions per Rule Type in SonarQube 50
5.7 Distribution of Priority for the individual Java Versions per Rule Type in SonarQube 51
5.8 Distribution of Java Versions Weighted By Method Occurences in % Across by Domain 52
5.9 Distribution of Methods based on Contributors for 0-25th Percentile 53
5.10 Distribution of Methods based on Contributors for 25-50th Percentile 53
5.11 Distribution of Methods based on Contributors for 50-75th Percentile 54
5.12 Distribution of Methods based on Contributors for 75-100th Percentile 54
5.13 Distribution of Methods based on LOC of Projects for 0-25th Percentile 55
5.14 Distribution of Methods based on LOC of Projects for 25-50th Percentile 55
5.15 Distribution of Methods based on LOC of Projects for 50-75th Percentile 56
5.16 Distribution of Methods based on LOC of Projects for 75-100th Percentile 56
5.17 Years of Actively Programming in Java by surveyed Software Developers 57
5.18 Most used Java Version in Projects by surveyed Software Developers 57
5.19 Reliance on Java built-in methods by surveyed Software Developers 58
5.20 Primary Domain of Work of Software Developers 59
5.21 Increase in Code Smells Detected by SonarQube After Adopting Java 8, 11, and 17 61
5.22 Increase in Bugs Detected by SonarQube After Adopting Java 8, 11, and 17 . . . 61

List of Tables

4.1 Overview of Data Sources and Tasks 26
4.2 Overview of the sample 27
4.3 Summary of the Entities and Methods from the Java Versions 29
4.4 Java Methods Analysis on GitHub 31
4.5 Overview of the SonarQube Rules 33
4.6 Overview of the Distribution of Project Domains 35
4.7 Overview of the Questions from the Survey 39

5.1 Java Built-in Methods Results Overview 42
5.2 Top 3 Most Used Methods by Version 45
5.3 Top 3 Least Used Methods by Version 46
5.4 Number of Bugs, Code Smells, and their Normalized Metrics 48
5.5 Most used Java built-in methods from Java 8, 11, and 17 by surveyed Software Developers 59
5.6 Least used Java built-in methods from Java 8, 11, and 17 by surveyed Software Developers 60

List of Algorithms

4.1 Java Class Processing Pseudocode 31

1 Introduction

In this chapter, we delve into the motivations behind our research. We are particularly interested in understanding how developers adopt new features introduced in various Java versions on platforms like GitHub. We are also interested to understand how these changes within the Java ecosystem impact software quality. To offer a comprehensive understanding, we have formulated specific research questions, which will be presented in the subsequent sections. Finally, we outline the thesis structure to guide the reader through our research.

1.1 Motivation

The software development landscape has changed dramatically since its origins, with programming languages continually adapting to meet the ever-growing demands of the industry. Java, which has had many major versions released since its introduction in 1996, is an example of this evolution [1]. Each version promises enhanced performance, augmented usability, and superior robustness, with the objective of addressing the challenges of contemporary software development. However, despite the introduction of a myriad of features across these iterations, there remains a notable gap in understanding the real-world adoption of these features and their subsequent impact on software quality.

Preliminary studies have highlighted the potential influence of specific Java features on their adoption within software projects [MAT06] [PBM12]. Yet, a comprehensive overview detailing the adoption patterns of various Java versions and their respective features across diverse open-source project types and domains is lacking. This void underscores the need for an in-depth study to better equip the software engineering community with insights into the adoption trends of Java features. In addition, understanding the specific benefits and challenges associated with each feature can provide a guide for future Java-based development projects.

One interesting trend in Java's history is the adoption of its Long-Term Support (LTS) versions. These versions are like milestones, offering more extended support than regular versions. For a long time, Java 8 was the most popular choice. But around 2022, things started to shift. "For many years, Java 8 has remained the leading Java version. However, its usage dropped by 12 percentage points in 2022. This is not surprising because, in March 2022, Java 8 lost its Oracle Premier Support. Thus, many developers decided to migrate their applications to different versions. The emergence of Java 17, a new long-term support version, attracted many users. Moreover, other versions like Java 11

[1]Java versions https://docs.oracle.com/javase/specs/jvms/se21/html/jvms-4.html#jvms-4.1-200-B.2

and Java 16 have gained greater audiences." [2]. Java 11 began to rise in popularity, and over the past three years, it has become the most used version. Even Java 17, a newer release, is seeing a surge in use. This data is derived from two separate surveys conducted by JetBrains[3] and NewRelic[4].

Despite the valuable insights that surveys like JetBrains' and NewRelic's provide, they primarily capture a snapshot of version adoption preferences. What remains unexplored is the practical impact of adopting features from these versions on software quality across different project types and domains, as well as the deeper patterns of feature usage across these projects. To address this gap, this research looks at Java projects to analyze the adoption of different Java versions and features in different open-source projects. Using the Mining Software Repository (MSR) technique, this study will examine a wide range of open-source projects on GitHub. To gain a broader perspective, a quantitative survey of developers was also conducted, providing valuable insight into their experiences and perceptions of Java feature adoption. Guided by research questions instead of predefined hypotheses, this study adopts an exploratory approach [RM16]. Using an extensive dataset, it aims to uncover patterns, correlations, and formulate new hypotheses for future empirical research.

1.2 Research Questions

In this research, our primary objective is to conduct a comprehensive study of the distribution of the various versions of Java and related features, which have been introduced over the years, in open-source projects. As a specific focus within these features, we have chosen to center our research on the adoption of Java methods introduced across the different versions. Through our in-depth analysis, we aim to understand which methods and features are gaining widespread acceptance and identify the domains in which they are predominantly used.

Our study analyzes not only the patterns used but also the effects of these patterns on software quality. We want to find out how the decision for a certain Java version and the associated Java features can affect the software quality of projects.
In addition, our research aims to identify key factors, such as a project's size, age, and the number of contributors who have worked on it, that drive the adoption of recent Java features and the different versions Java offers these days.
By contrasting our numerical results with quantitative feedback from developers, we aim to correlate the trends we have observed in the adoption of Java features in open-source projects with the first-hand experience of Java experts.

For this purpose, we have formulated four main research questions (RQs). Specifically, the first RQ is further broken down into four sub-questions to provide a comprehensive understanding of its scope.

RQ1 How does the adoption of new Java version features manifest on GitHub?

RQ1.1 What is the degree of adoption of new Java version methods?

[2]JetBrains Developer Ecosystem `https://www.jetbrains.com/lp/devecosystem-2022/java/`
[3]JetBrains `https://www.jetbrains.com`
[4]NewRelic `https://newrelic.com/resources/report/2022-state-of-java-ecosystem`

- **RQ1.2** Which methods, introduced in various Java versions, are most frequently adopted in GitHub repositories?
- **RQ1.3** Which methods, introduced in various Java versions, are least frequently adopted in GitHub repositories?
- **RQ1.4** How has the adoption rate of new Java built-in methods evolved on GitHub from 2011 to 2023?

RQ2 How is the adoption of Java version features related to software quality?

RQ3 What is the impact of domain distribution, contributor count, and project size on the adoption of new Java version features on GitHub?

RQ4 How do the observed results for the adoption of Java version features align with the personal experiences of Java developers?

In summary, this research aims to provide an understanding of the distribution of different versions of Java and their built-in methods, but also of developers' choices in adopting versions of Java and the related features. It also aims to show the far-reaching consequences of these decisions on software quality.

1.3 Structure of the Thesis

The thesis consists of eight chapters, beginning with an introduction. In the second chapter, we define the basic information and provide the reader with the necessary background knowledge to understand the core concepts of our research. In the third chapter, we review previous studies that have both influenced and inspired our research direction. The fourth chapter takes an in-depth look at our research methodology, explaining the subject matter of our study, the sample selection criteria, the data collected, and the step-by-step approach we took. Subsequently, in the fifth chapter, we present our main findings, which directly answer the research questions we formulated. In the sixth chapter, we interpret our results and discuss the possible implications. In the seventh chapter, we critically examine the extent to which the validity of our research is compromised. Finally, the eighth chapter, which concludes the thesis, briefly summarizes the essence of our work and previews future work that can be done.

2 Background

This chapter covers the Java programming language and the concept of software quality. We will begin by diving into Java's journey, exploring its various versions released over the years. Special attention will be given to the Long-Term Support (LTS) versions, which have played an important role in Java's evolution and widespread adoption. Following that, we will discuss what software quality means, its significance, and why it is important for any software project, with regard to bugs and code smells.

2.1 Java

Java, a programming language developed in the 1990s, is the brainchild of James Gosling, Mike Sheridan, and Patrick Naughton. The first version was released by Sun Microsystems in 1996. One of its outstanding features was the ability to write software that could run on different computers and devices without modification, a principle often described as "Write Once, Run Anywhere.". This principle was made possible by the Java Virtual Machine (JVM), which allows Java software to be executed consistently across various hardware platforms.

In addition, Java emphasized security and created a protected environment for running its applications to reduce the risk of malicious attacks. It also provided tools that made it easier for developers to write software that could perform multiple tasks simultaneously. With an extensive library of pre-built code and an approach that promotes writing code in an organized and reusable manner, Java made the development process more efficient. It also introduced features that automatically managed a program's memory usage and freed developers from some of the more tedious aspects of programming. These combined strengths contributed greatly to Java's acceptance in the tech industry.

Until 2018, this programming language has grown and expanded dramatically, becoming a few years after its introduction, one of the most widely used and trending languages of the moment. In 2018, Java was displaced from its long-standing position at the top of many language rankings. However, it remains a leading and influential language through 2023, having released 21 versions over its lifetime.
Over the years, the various Java versions have added to Java's success by improving performance, introducing innovative features, and fixing bugs and vulnerabilities. LTS versions have played a crucial role in providing stability and extended support for enterprise applications. These versions have been supported for at least 5 years and have provided a reliable foundation for critical software development. Overall, this approach has kept Java relevant and widely used until 2023, contributing to its persistent reputation and importance in the IT industry.

Java has seen numerous updates since its inception, but a few versions stand out for their significant impact on the software development landscape.

Released in March 2014, Java 8 [1] was revolutionary for integrating functional programming paradigms into the language. It introduced Lambda Expressions, enabling more concise and readable code, and the Stream API, which transformed the way developers processed sequences like collections. Moreover, with the introduction of new date-time classes and the Nashorn Engine [2], Java 8 provided tools for more diverse and efficient software creation.

Following Java 8, Java 11 [3], released in September 2018, further streamlined the developer experience. The ability to execute Java programs directly from source code using the Java command made development and testing quicker. This version also modernized web interactions by offering a new HTTP client that supported the contemporary HTTP/2 and WebSockets protocols. As an LTS release, Java 11 assured enterprises of extended stability and support.

Then in September 2021, Java 17 [4] continued the trend of substantial improvements. It improved random number generation processes, introduced a new rendering pipeline for macOS, and enhanced support for Unix-domain socket channels. Much like Java 11, this release was marked as an LTS, emphasizing its role in providing a dependable foundation for large-scale applications.

The LTS designation for versions 8, 11, and 17 underscored Java's commitment to catering to enterprise needs. While frequent updates are great for introducing new features, enterprises crave stability and long-term support. LTS releases ensured that businesses had a robust and reliable platform for years on end.

All the information about the programming language and the features implemented into the various versions were retrieved from the official documentation provided by Oracle [5].

2.2 Software Quality

To establish the concept of software quality, we must first provide a brief description of what is quality. In the book "Software Product Quality Control" by Wagner [Wag13], the term "quality" is analyzed. Wagner explores the philosophical roots of the term and notes that "quality" has often been equated with "good" for over 2,000 years. This association underscores the continuing importance of the concept, particularly in the field of software, where quality often determines perceived value. Software quality refers to the degree to which software meets user expectations and provides desired functionality in a reliable and efficient manner, and in Wagner's book we find the description of quality models that have been developed over the years to define and measure software quality. One of the renowned models is ISO/IEC 9126 [6], which has been replaced by ISO/IEC 25010 [7]. These models divide software quality into specific factors such as functional suitability, reliability, performance efficiency, usability, security, maintainability, portability, and compatibility. Each factor provides a unique perspective to evaluate the effectiveness of the software in fulfilling its intended purpose.

[1] Java 8 https://docs.oracle.com/javase/8/docs/
[2] Nashorn Engine https://docs.oracle.com/javase/10/nashorn/introduction.htm#JSNUG136
[3] Java 11 https://docs.oracle.com/en/java/javase/11/
[4] Java 17 https://docs.oracle.com/en/java/javase/17/
[5] Oracle Java Documentation https://docs.oracle.com/javase/specs/index.html
[6] ISO/IEC 9126 https://de.wikipedia.org/wiki/ISO/IEC_9126
[7] ISO/IEC 25010 https://iso25000.com/index.php/en/iso-25000-standards/iso-25010

Our focus in this research is on the reliability and maintainability aspect of the software quality defined in the ISO/IEC 25010 model. Reliability in this model refers to how consistently a system functions as intended under specific conditions. This includes the system's operational availability, its ability to function despite errors, and its ability to recover data and return to a desired state after interruptions. Meanwhile, maintainability indicates how efficiently and effectively a product or system can be changed, whether to improve, correct, or adapt to changing conditions.
These two aspects are relevant to our research, especially because of their relationship with code smell and bugs. Some studies have examined how code smell and bugs can affect the reliability and maintainability of a system [YM12a][SYA+13] [SMLG17].

3 Related Work

Several studies have previously examined the adoption and impact of various Java features to some extent. However, it is noteworthy that there is a lack of research focusing on the more recent Java versions, especially the LTS ones, and the specific adoption patterns of Java methods. While these studies may be limited, they provide valuable insights into research methodologies and the general approach taken. Since our research is in the direction of data mining, we also considered studies that used data mining techniques on GitHub and integrated SonarQube. This chapter focuses on the existing research that guided and influenced our study.

In the 2006 study “Usage Patterns of the Java Standard API” [MAT06] Ma and his team looked at something similar to our research. In this study, they utilized a corpus of open-source Java software to investigate the extent to which various features of the Java Standard API are adopted. Their methodology involved classifying Java entities from the API used within a Java source code file into types, classes, methods, or packages. They used the Jepends tool, originally developed for identifying refactoring candidates, modified to record potential API uses in the source code.
The study found that only about 52.0% of classes within the Java 1.4.2 API were actively used, and about 21.0% of methods were used. The main reason for this study was to see that even though the Java API has many features, many of them are not used in regular coding. The paper takes a closer look at the most commonly used entities within the API, noting some expected and unexpected trends. For example, the high frequency of the “Exception” class is a notable observation. The results of the study raise questions regarding the representativeness of the corpus and the possible need to refine the API to better meet the needs of developers.
In essence,this study underscored the fact that a large portion of the Java API often goes unused in regular development.

The second relevant study was “Adoption and Use of Java Generics” [PBM12] by Parnin et al. In this study, they investigated the use of Java generics in software development. Using a custom algorithm, they analyzed large codebases to identify patterns of Java generic usage. Their methodology, rooted in the examination of the Abstract Syntax Tree (AST), provided a deep dive into how this Java feature is integrated into open-source projects.
The study underscored a pronounced migration from raw types to generics, indicating an emerging penchant for type-safe programming in Java - an approach that prioritizes type error bypassing and strengthens code reliability. Still, the study was not without challenges. In particular, they struggled with detecting “implicit” parameterized types in nested method calls, leading to possible underrepresentation in their data.
Interestingly, certain results diverged from preliminary assumptions, underscoring the fluid and sometimes unpredictable nature of software development methods. In essence, Parnin’s study combines rigorous methodology with pragmatic insights, offering clarity on the nuances and evolving trends in the adoption of Java generics in software development.

In a comprehensive study of the Java Standard API [ANMT11], Anslow et al. (2010) emphasized the value of software consisting of smaller packages and classes to simplify maintenance. They pointed out the substantial growth of the Java Standard API, which now includes over 200 packages, about 5,800 classes, and about 50,000 methods.
Using visual software analysis, they applied established software visualization techniques to identify the API's large packages and classes. These results pointed to potential refactoring areas. The research was anchored in the Java Standard API 6 and the Qualitas Corpus, a set of 100 open-source Java applications, to examine software structure and design quality characteristics. They combined software metrics with visualization approaches such as the System Hotspot View metaphor to show the size and complexity of classes and packages. In particular, the java.awt and javax.swing packages turned out to be some of the largest in the Java Standard API 6.
An important observation was that abstract and concrete classes in the Java Standard API had higher Weighted Methods per Class (WMC) and Lines of Code (LOC) than Java software in the Qualitas Corpus.
Overall, the study found no correlation between usage and the size of Java API packages and classes.

In 2016, Qiu et al. presented a comprehensive analysis in their paper titled "Understanding the API usage in Java" [QLL16]. Their primary aim was to discern patterns of API usage in real-world Java projects. They analyzed over 5000 open-source Java projects, amounting to more than 150 million source lines of code. To ensure accurate data extraction, they developed and utilized a tool named "Java API Usage Extractor (JAPIExtractor)". This tool was responsible for collecting, managing, and analyzing API usage from the large corpus.
For their analysis, they introduced specific metrics to evaluate the frequency of API usage. One such metric is the "Simple Occurrence" (SO), which counts the direct usage of an API entity without considering its related components. On the other hand, the "Cumulative Occurrence" (CO) metric takes into account both the direct usage of an API entity and the usage of its related components. Using Maven, a prevalent build management system, they were able to automatically resolve dependencies, generating resolved abstract syntax trees (ASTs) from the source code for precise type bindings of API entities.
Their analysis spanned both core and third-party API libraries. Among the significant findings was the observation that API usage adheres to the Zipf distribution, and a considerable portion of the core API remains unused in real-world scenarios. Alarmingly, they also discovered that deprecated API entities, even those marked obsolete for extended periods, were still actively used in many projects. Their findings provide good insights into actual API usage patterns, suggesting potential improvements in language API design, teaching methodologies, and library recommendations.

Another study from 2020 that analyzed open-source projects on GitHub was that of Wang et al. in their paper "An Empirical Study of Usages, Updates and Risks of Third-Party Libraries in Java Projects" [WCH+20]. In this paper the authors conducted an empirical study of Java open-source projects on GitHub. They selected projects with over 200 stars to ensure quality and relevance. The goal of the study was to understand the dynamics of third-party library usage, updates, and risks. The authors collected data on library dependencies, API calls, version releases, and bugs. They analyzed usage trends, version popularity, and reasons for updating or avoiding updates. A key component of their approach was quantifying the risks associated with outdated libraries and delays in updates.
They also examined developer responses to faulty library versions and assessed the need for an outdated library alert system. It was found that these projects rely heavily on third-party libraries,

with an average project involving 63.5 libraries. However, there is a discernible delay in adopting the latest versions, with projects typically using libraries that are 3.2 versions behind the latest version.
Although projects and libraries are updated frequently, the incorporation of these updates is significantly delayed (142.8 days on average). The study also highlighted the risks associated with using outdated library versions and examined how developers react to faulty versions.
These findings underscore the challenges developers face and highlight the need for efficient warning systems and automatic updating of outdated libraries. To facilitate this, the authors presented a prototype error-driven alert system that helps developers make informed decisions about updating library versions.

One study that examined a particular Java method was by Sulír,"String Representations of Java Objects:An Empirical Study" [Sul20]. The purpose of this study was to investigate the extent to which classes define their own toString methods rather than relying on the default representation. Considering type hierarchies, the study highlighted that while a superclass may provide a toString definition, it does not have the specificity that a class-defined method would provide. Rather than sourcing projects from GitHub, the researchers analyzed a corpus of parsable source code files, using the Spoon library for source code automation. The results showed that the vast majority, 73.0%, of classes relied on the default toString representation of the Object class, which often lacks descriptive utility. In addition, 63.0% of toString calls are implicit, primarily through string concatenation with a non-string operand.
In the future, the researchers plan to expand their study and use a larger and more representative code corpus. They are particularly interested in delving into aspects of the nature and structure of toString methods.

The last study, which performed exhaustive data mining research analyzing the impact of language features on software quality, is from Ray et al., "A Large Scale Study of Programming Languages and Code Quality in Github" [RPDF17].
In this study, they tried to empirically determine the impact of programming languages on software quality. Using an extensive dataset from GitHub that included 729 projects, 80 million lines of source code (SLOC), contributions from 29,000 authors, 1.5 million commits, and 17 languages, the authors used a mixed methods approach. This approach combined multiple regression modeling with visualization and text analysis to assess the impact of language features, such as static vs. dynamic typing and strong vs. weak typing, on software quality.
Results suggest that while language design exerts a significant, albeit modest, impact on software quality, factors such as project size, team size, and commit size outweigh this impact. In particular, it was found that strong typing appears to be slightly superior to weak typing. Among functional languages, static typing also appears to have an advantage over dynamic typing. However, they caution that these modest effects may be due to other intangible factors, such as the propensity of certain personality types to use functional, static, and strongly typed languages.

Each article we cited gave us good insights into the process of data collection and analysis. They gave us guidance on where to find open-source projects and how to develop a methodology to obtain the data we needed for our research. In particular, the studies by Ma et al. [MAT06] and Anslow et al. [ANMT11] aligned with our objective. However, they were not as specific as our study in their research questions and focused on a specific, older version. These studies date back to 2006 and 2010, respectively. Despite their age, the methodology and process are still interesting and offer several insights.

4 Methodology

In this chapter, we describe the methodology used to gather, process, and analyze the data for our research. The methodology consists of five main steps that were needed to implement in order to get the necessary information to answer the research questions. These five steps are:

1. **GitHub data mining:** We used the GitHub API [1] to mine relevant repositories and their data for subsequent analysis.
2. **Oracle documentation data scraping:** We used the BeautifulSoup library [2] to scrape relevant information about methods and their versions from the Oracle documentation about Java.
3. **Java class processing:** We used the javaparser library [3] to extract information about the structure, dependencies, and methods of the Java classes.
4. **SonarQube analysis:** We used the SonarQube API [4] to categorize and filter the rules for Java and evaluate the code quality of the Java code.
5. **Survey:** We conducted a survey with different Java developers to gather information on the adoption of Java versions and the related features used in the industry.

We describe each of these steps in more detail in the following sections. Table 4.1 provides a summary of the five main tasks were implemented in our research, including the objective, the method, and data retrieved for each task. All data used for this research, along with the processing script that creates and analyzes this data, can be found in the GitHub repository [5].

4.1 Study Object and Sampling

This research focuses on the Java programming language, particularly on the built-in methods introduced across various versions. The main objective of this research is to understand the adoption and utilization of these methods in contemporary software projects. Instead of relying on older corpora, which may be assembled for different purposes, could be outdated, and potentially carry

[1]GitHub API https://docs.github.com/en/rest
[2]BeautifulSoup https://www.crummy.com/software/BeautifulSoup/
[3]javaparser, https://javaparser.org
[4]SonarQube API https://docs.sonarsource.com/sonarqube/latest/extension-guide/web-api/
[5]Research Data Repository https://github.com/danielabajirov/Exploring-the-Adoption-of-Java-Version-Features-and-Their-Relationship-to-Software-Quality-on-GitHub

Task	Objective	Data Source/Method	Outcome
GitHub Mining	Extract Repository Data	GitHub's API	Repository Data: code, Docs, Commits
Oracle Doc Scraping	Extract Java method Info	BeautifulSoup	Java Methods: URL, Name, Version
Java Class Analysis	Extract Java Class Structure	javaparser library	Relevant Method and Class Data
Code Quality Analysis	Software Quality	SonarQube API	Code Quality Metrics: Bugs, Smells
Developer Survey	Validate Results	Survey with Java Developers	Distribution of Java Versions and Associated Methods in the Industry

Table 4.1: Overview of Data Sources and Tasks

biases from their original research goals, we opted to directly mine software repositories to gain a more current and unbiased understanding of Java method adoption.

Historically, several tools like Softchange[6] and Hipikat[7] have been recommended for data mining repository studies. However, many of today's prominent online software development platforms, where repositories are hosted, offer their own APIs for data access. Such APIs are straightforward to implement, providing greater flexibility with data filtering and other custom operations to meet specific research needs.

Given these advantages, we chose GitHub [8] as our primary data source. As one of the most extensive and frequently updated platforms, GitHub offers a rich snapshot of contemporary software development trends. Numerous academic studies use GitHub for their research and data collection [GGTF15][CS15], which makes our choice even more credible. By extracting data directly from GitHub, we ensure that our results reflect the most current, comprehensive, and relevant data, providing an authentic representation of Java method adoption in current software projects.

To obtain a valid set of projects for analysis, we began with a data refinement process that proceeded in three specific phases and excluded projects that did not meet our established criteria. Heeding the insights from Kalliamvakou et al.'s research [KGB+15], we set criteria: a project must have at least 100 stars, more than 6 commits, and at least one commit in the past six months.

The refinement process can be summarized in the following stages:

1. First, we identified the projects based on the number of stars using the GitHub API and the Python library requests [9]. We then ensured that each project primarily used Java, with Java accounting for more than 75.0% of the codebase.

[6]Softchange https://sourcechange.sourceforge.net
[7]Hipikat https://www.cs.ubc.ca/labs/spl/projects/hipikat/
[8]GitHub https://github.com
[9]requests https://pypi.org/project/requests/

2. Next, by examining the commits information, we verified if the repository had been active within the last six months.
3. Finally, we fetched and stored metadata about each project locally. This data included details such as the number of contributors, issues, and commits each project had.

After this refinement process, we ended up with a sample of 2188 projects with 41,543 contributors and more than 12 million commits.

Projects	**Contributors**	**Commits**	**Average Project Stars**
2188	41,543	12,074,878	496.15

Table 4.2: Overview of the sample

4.2 Data Collection

As shown in Table 4.1, we have five types of data sources from which we gathered our data. To retrieve data from these sources, each required a unique implementation. In this section we have grouped together two processes, Oracle documentation and Java class processing. The data sets obtained from these two processes can be used as input for subsequent processes, which will be explained in more detail later.

4.2.1 Oracle Documentation

For our research, we need information about when certain Java built-in methods were introduced in specific versions. Unfortunately, we have not found any existing research or precompiled dataset (e.g., CSV or XML) that allows us to easily associate these methods with the versions in which they were introduced. A potential source for obtaining this type of information is the Oracle official documentation of Java, where we can find the documentation for Java 8 [10], 11 [11] and 17 [12] respectively. Oracle's official Java documentation provides a comprehensive overview of the Java programming language and its libraries. Within the documentation of each version, there is an outline, a list of packages, and detailed descriptions of classes/interfaces, highlighting their intent, associated methods, and annotations. Importantly, these method descriptions incorporate a **Since:** label that indicates the version in which they were introduced. Using the information contained in this label, we can systematically assign each method to the version in which it was introduced. To eliminate the need for manual extraction and categorization, we developed a web scraping script that simplifies and automates the entire process.

To facilitate the implementation of the web scraping script, we used the Python library BeautifulSoup. Beautiful Soup is a tool utilized for extracting data from HTML and XML files, commonly employed in web scraping tasks. It constructs parse trees from HTML and XML, facilitating easy data retrieval. The library offers several straightforward methods and idioms, inherent to Python, for traversing,

[10] Java 8 Documentation, https://docs.oracle.com/javase/8/docs/api/
[11] Java 11 Documentation, https://docs.oracle.com/en/java/javase/11/docs/api/index-files/index-1.html
[12] Java 17 Documentation, https://docs.oracle.com/en/java/javase/17/docs/api/index-files/index-1.html

exploring, and altering a parse tree. BeautifulSoup was used in navigating through the Oracle's documentation, enabling the creation of mappings between methods and their versions. As already mentioned, each Java class includes various elements, especially the associated methods. To create a comprehensive mapping between methods and their respective versions, it is important to first obtain a complete list of all Java classes. This then facilitates the extraction of the corresponding methods. Therefore, the implementation strategy for deriving the method mappings has been divided into two distinct phases. Below, we describe the two-phase approach that culminated in the final mapping of the methods to their corresponding versions.

Phase 1: In Oracle's official documentation for each Java version, there is a dedicated index section. This section lists all classes, interfaces, methods, and other entities introduced in that specific version and its predecessors. For example, consulting the documentation for Java 11 will present entities introduced up to and including version 11, but it will exclude those from subsequent versions.

In the implementation process, the initial step involved visiting the index associated with each Java version. Subsequently, data was extracted from these indices, categorizing entities based on three attributes: Entity, Description, and URL. This structured categorization facilitated cross-validation, ensuring consistency and accuracy in the data extraction and subsequent mapping. It prevented inconsistencies, such as a method documented in Java 11 being erroneously mapped to Java 8 when it was not present in the latter's documentation.

The scraping process relies on specific HTML tags to automate data retrieval. However, modifications were necessary when transitioning from scraping information from Java 8 to Java 11 and 17. The Oracle documentation introduced a new page format for these versions, resulting in changes to the HTML tags. The first notable difference was the URL leading to the index. The second significant change was the removal of the **<span/>** tag, which had been used to encapsulate the required information. In the later versions, the information was solely contained within the **<a/>** tag.

Phase 2: Using the data from the first phase, we used the methods and their corresponding URLs to determine their versions and other pertinent details, which further facilitated the subsequent mapping process later on. For this phase, we developed a script that facilitates iteration through each URL and extraction of the version relevant to the method. During this implementation, we identified three distinct cases which necessitated unique handling:

1. The description section of a method includes a **Since:** label. This label denotes the version in which the method was introduced. When present, we directly scraped this information, aligning it with the respective method.

2. Absence of a **Since:** label in the method's description. Under such circumstances, we referenced the parent class of the method in question. Typically, the class description contains the **Since:** label, providing the requisite information.

3. Neither the method nor its parent class exhibits a **Since:** label. Through careful manual analysis, we have found that in the cases where both the method and the class are unmarked, they were introduced in the 1.0 version of Java. To confirm this observation, we manually checked 50 such cases. Our results consistently showed that the associated class was introduced in Java 1.0.

To increase the precision of the later mapping process, we also extracted additional metadata, such as details about parameters, associated classes, and the hierarchical structure of the classes involved. This additional information played an important role in refining our mapping process and enabled us to make clearer connections between Java methods and their specific versions.

For a comprehensive overview of the entities and methods corresponding to specific Java versions, we can take a look at Table 4.5. The table illustrates the increase in entities and methods from Java 8 to Java 11 was more modest than the growth observed between Java 11 and Java 17.

	Java 8	**Java 11**	**Java 17**
Number of Entities	51,190	52,417	55,440
Number of Methods	34,676	34,916	37,584

Table 4.3: Summary of the Entities and Methods from the Java Versions

4.2.2 Java Class Processing

Our sample consists of projects primarily written in the Java programming language. The projects in our sample use various Java versions. Each version introduces a range of new features, including the addition of new classes and methods, or extensions to existing ones. We have chosen to center our research on the adoption of Java methods introduced across various versions. For this reason, we must analyze these projects to extract relevant information that addresses our research questions. Therefore, the analysis of these Java projects necessitates a tool that can extract detailed information about the methods, their usage, and other associated information. By "associated information" about a method, we refer to details like its parameters, return type, access modifiers and the possible libraries that a class has.

For our analysis, we considered three different options. The first option was to utilize an existing tool, JAPIExtractor, implemented by Qiu et al. in their research "Understanding the API usage in Java" [QLL16]. JAPIExtractor is a tool for analyzing and cataloging Java API usages from a given corpus. Leveraging Eclipse integrations, Maven API and advanced parsing techniques, it provides detailed insights into API dependencies, deprecated API entities and offers statistical analysis capabilities.
However not only were we unable to locate the source code of this tool, but even if we had, its original scope would have necessitated modifications to extract information pertinent to our specific research objectives.

Without access to the source code of JAPIExtractor, another option was to create a similar tool from scratch. We could proceed similar to how Qiu et al. proceeded, and this would necessitate using the Eclipse JDT parser to parse Java code and subsequently extract method information from its abstract syntax tree. However, building such a tool introduces the risk of obtaining unreliable results if the analysis is incorrectly implemented. Given the complexities and potential problems of this approach, we opted for the third option: using the javaparser library. Javaparser is an open-source

Java library that parses Java source code into an abstract syntax tree (AST). It enables developers to programmatically parse, modify, and generate Java code, facilitating tasks ranging from user-defined static analysis to code transformation and generation.

We chose the javaparser library because of its several advantages. First, as an established tool, it offered significant time savings that supported our development process. Such mature libraries have extensive documentation and are further developed by the community, which ensures robustness and functionality. Developing a custom parsing solution offers more control and customization, but is often less robust and feature-rich compared to proven libraries. For our needs, javaparser was suitable, providing efficient extraction of method details and metadata from Java classes.

The implementation is divided into two main components. The first component constitutes the core of the script, focusing on the analysis of the repository and the extraction of necessary data, which is subsequently saved into a file. The second component augments the implementation with the ability to execute git commands. This allows for the repository's history to be reset to a specific date, followed by the analysis. Subsequently, the repository's history is reverted to its original state.

To better understand the process and how the two parts were implemented, we can take a look at the Algorithm 4.1. The algorithm uses two main inputs: "inputDirectory", which is a list of where the projects are stored, and "inputDatesCommit", a list of dates to check in the project's history. These dates help in resetting a project's git history. If provided, we iterate over each repository and date. If a commit is found on a specified date, the project's history is reset to that date. If not, we check each preceding day until a commit is found or until we reach January 1st of that year. During this iteration, if a valid date is not identified or an error arises, we record the necessary details for later review.
After we successfully reset a project to a certain commit, we can proceed with the analysis. We iterate through each Java class file in the project and provide them as input to our core analysis method. This method, utilizing the javaparser library, then extracts the essential information from the class, including methods, their usage, and the other specific details we outlined earlier. After completing the extraction, we store the data in a CSV file. Alongside the extracted information, we also annotate the file with additional details such as the class and the project analyzed. We repeat this process for each project. Once all projects have been analyzed, the iteration concludes.
Upon completing the analysis for a particular date, the repository's history is restored to its latest commit. Any issues encountered during these phases are logged, and an error count is updated.
During the analysis, we encountered 21 projects which resulted in a "StackOverflowException" and could not be parsed. Consequently, we chose to exclude these projects from our sample and subsequent analysis. Details about these projects, along with descriptions of the encountered errors, are documented in our repository.

We extracted a dataset of about 85 million methods from a wide range of projects with a total of 151,185,183 lines of code and 85,887,603 total number of methods, as shown in Table 4.4. Excluding the standard setters and getters resulted in method counts of 58,887,603 and 105,285,183 lines of code, respectively. On average, the data showed about 568 methods used per KLOC.

Metric	Value
Total Lines of Code (LOC)	151,185,183
Total Lines of Code Without Set/Get (LOC)	105.285.183
Total Numbers of Methods	85,887,864
Total Numbers of Methods Without Set/Get	58,887,603
Methods per KLOC (Thousands of Lines of Code)	568.21
Methods per KLOC (Thousands of Lines of Code) without Setters/Getters	559.5

Table 4.4: Java Methods Analysis on GitHub

Algorithm 4.1 Java Class Processing Pseudocode

```

startAnalysis(inputDirectory, inputDatesCommit){
    initializeLogger();
    initializeValuesForErrorHandling();
    if(inputDatesCommit){
        resetRepositoryToValidDate(inputDatesCommit);
        analyzeRepository(inputDirectory);
        writeInformationToCsv(analysisOutput);
        revertRepositoryOriginalState();
    } else {
        analyzeRepository(inputDirectory);
        writeInformationToCsv(analysisOutput);
    }
}

analyzeRepository(inputDirectory){
    for File in inputDirectory{
        getDataImports(File);
        getDataAnnotations(File);
        getMethodInformation(File);
    }
}

```

4.3 SonarQube Analysis

The second research question of our study is related to software quality, especially code smells and bugs. For this reason, we need to analyze our sample using a static analysis tool that can provide us with information about these two aspects in each project. Our goal is to find specific code smells and bugs related to a particular Java version. For example, with the introduction of new features, an older one may be deprecated and need to be changed, resulting in code smells, or a new feature may fix a bug in an older one.

Today, numerous tools are available for static analysis of Java code to identify code smells and

bugs. Notable examples include SpotBugs [13], PMD Java [14], Spoon [15], SonarQube [16], among others. For our research, we chose SonarQube. Numerous studies have employed SonarQube [QGB+17][SBLR19] for software quality analysis, demonstrating its versatility and the wide range of metrics it offers.

SonarQube is an open source tool for continuous code quality review. It automatically identifies potential problems such as bugs, vulnerabilities and code smells that deviate from best practices. Originally focused on Java, SonarQube now supports over 20 programming languages. Its straightforward local deployment makes it particularly suitable for research projects where consistent code quality checking is part of the scope. Another factor we considered when choosing this tool is that, for the Java programming language, SonarQube offers the most extensive support in terms of defined rules compared to other tools.

Our decision to utilize SonarQube was further reinforced by its vast collection of Java-specific rules. With a total of 622 rules, Java is the most integrated programming language in SonarQube. For our research, we did not require the full set of rules. We were particularly interested in rules specific to certain Java versions. This selection process was done manually, due to the complexity of the task and the inaccuracy, that an implementation to automate the categorization could have. SonarQube's rules are not directly version-specific, but their detailed descriptions sometimes allow conclusions about the Java version they refer to. Our manual selection resulted in 72 rules: 46 related to code smells, 24 to bugs and 2 to vulnerabilities.

After the initial filtering, we used the rule set to perform an in-depth analysis of the locally stored repositories. This step required a preconfigured script to start the analysis for each repository and save the results locally. Figure 4.1 provides a comprehensive overview of the architecture of the automated analysis. It starts with the manual categorization of Java rules from SonarQube and the repositories and ends with the data relevant for the analysis, especially code smells and bugs.

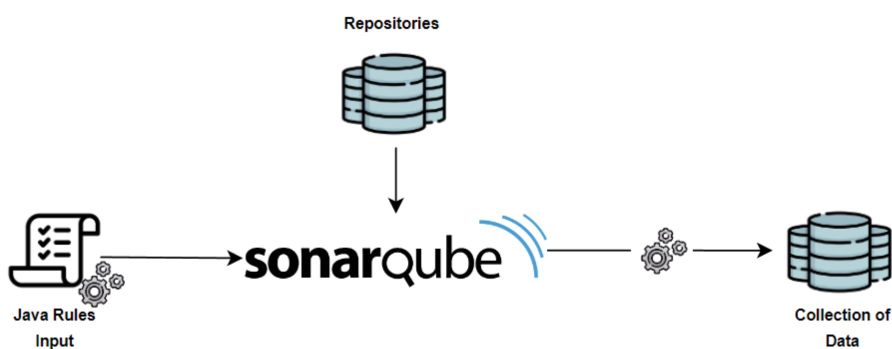

Figure 4.1: Process Overview: Collecting Java Methods Software Quality Data

[13]SpotBugs https://spotbugs.github.io
[14]PMD Java https://pmd.github.io
[15]Spoon https://spoon.gforge.inria.fr
[16]SonarQube https://www.sonarsource.com/products/sonarqube/

Total Java Rules	Specific Rules to Java Version	Specific Rules to Type Bug	Specific Rules Type Code Smell	Specific Rules Type Vulnerability
622	72	24	46	2

Table 4.5: Overview of the SonarQube Rules

4.4 Identifying Project Domains

Several factors may contribute to the adoption of new Java versions in GitHub projects. Among these factors, the domain of the project is of great interest to our research. In this context, “domain” refers to the specific area or category of application that a project falls under, such as web development, data analysis, mobile app development, or game development, among others.

By examining domains, we aim to identify potential differences in motivations and challenges that might influence Java built-in methods adoption. For example, a project in the game development domain might adopt a new Java version more rapidly if it offers improved graphics performance, while a project in the data analysis domain might prioritize versions that offer better data processing capabilities.

Understanding these domains may allow us to gain insights into why certain areas might approach Java feature adoption differently. This perspective should provide a deeper understanding of the factors that influence Java built-in methods adoption across different types of projects, ultimately improving the depth and insights of our research.

When determining the specific domains for categorization, we did not create these domain names ourselves. Instead, we adopted the domain names from a pivotal study by Borges et al. [BHV16] greatly influenced our decision. In their research, they manually categorized over 2,000 GitHub repositories, resulting in the identification of six unique domains: Application Software, System Software, Web Libraries and Frameworks, Non-web Libraries and Frameworks, Software Tools, and Documentation. The categorization method was based on a detailed analysis of the repository description, the GitHub page, and the project’s official page.

We adopted the domains from Borges et al. due to their more encompassing and detailed nature. However, our approach to categorization diverged. Manually categorizing projects is a labor-intensive task. Though feasible with a team, it becomes prohibitively time-consuming for an individual. Consequently, we aimed to automate the categorization process.

Some studies, like that of Dien et al. [DLT19], have shown that text categorization into distinct topics using Natural Language Processing (NLP) [17] techniques can achieve impressive accuracy rates of up to 91.0%, but still the implementation process can be quite complicated and time consuming.

While NLP techniques show promise in text categorization, applying them to categorize projects based on the extensive descriptions of READMEs presents unique challenges. READMEs vary significantly in structure and content, ranging from concise summaries to detailed technical expositions. This diversity can complicate the extraction of consistent features for categorization. Additionally, projects often use domain-specific terms, which might not be universally recognized

[17] NLP https://www.ibm.com/topics/natural-language-processing

by standard NLP models without specialized fine-tuning. Moreover, the overlap of terms across multiple domains in a single README can introduce ambiguities, making it challenging to discern a project's primary domain. Given the intricacies involved and our desire for a transparent and efficient methodology, we opted for an alternative approach.

Traditional NLP techniques, though effective for text categorization, often necessitate domain-specific training data, fine-tuning, and intricate feature engineering to cater to specialized tasks, such as categorizing diverse READMEs. Addressing these challenges requires a more versatile solution. Currently, several models are pre-configured and ready for application. The GPT-3.5-turbo model [18] from OpenAi [19] is a big name this year, and it has been showing great results in many studies. In particular, several recent studies have highlighted its accuracy and effectiveness in tasks such as categorization, further validating its usefulness [AAA+23] [KWNA23] [Lam23]. Its generative capabilities allow it to understand and generate human-like text, making it better suited for comprehending varied and domain-specific content found in project descriptions. Moreover, the sheer scale of its training data means it has likely encountered a wide range of terminologies across different domains, potentially minimizing the challenges associated with domain-specific terms. One of the compelling reasons we chose the GPT 3.5 Turbo model was the initial free tokens provided to each account, allowing us to experiment without incurring any immediate costs. For our comprehensive analysis of the 2166 projects, we only used $3.6 of the $10 credit offered. Using a model with initial free tokens and being transparent about our expenses also provides a clear financial roadmap for future researchers who want to replicate or build upon our study.

The primary source of data for categorization was the project descriptions. To prepare these descriptions effectively as input for the model, we initiated a filtering and cleaning process. Since many of these descriptions are written in markdown format[20], it was essential to preprocess them, removing any irrelevant tags or "noise" that might hamper the categorization. The GPT-3.5-turbo model imposes a token limit of 4096 per input. To provide context, a token can range from a single character to a whole word. For a rough approximation, this token count often translates to about 800-1000 words of plain text. If a project's description exceeded this limit after cleaning, we recorded the project's details in a log file and skipped it. All skipped projects were then manually reviewed to ensure we did not miss critical categorization opportunities. or validation purposes, we manually categorized a sample of 220 projects, which represented 10.15% of our dataset, and compared these categorizations to the model's outputs. The results showed that the GPT-3.5-turbo model's categorization diverged from our manual categorization in only 7 out of the 220 cases. This corresponds to an accuracy of 96.82%.

Table 4.6 shows the distribution of project domains we obtained after categorization. An even distribution is not given, as we have domains like "Non-Web Libraries/Frameworks" with 27.3% and on the other hand domains like "System Software" and "Software Tools" with only 10.0% and 9.0% respectively.

[18]GPT-3.5-turbo model https://openai.com/blog/gpt-3-5-turbo-fine-tuning-and-api-updates
[19]OpenAi https://openai.com/
[20]Markdown https://markdown.de

Domain	Number of project	% of project
Non-Web Libraries/Framework	591	27.3%
Web Libraries/Framework	440	20.3 %
Documentation	409	18.9%
Application Software	316	14.5%
System Software	216	10.0%
Software Tools	195	9.0%

Table 4.6: Overview of the Distribution of Project Domains

4.5 Mapping Process

At the end of data collection, we ended up with a significant volume of data. In order to draw meaningful insights and directly answer our research questions, this data had to be refined and filtered.
A crucial step in this process was the integration of the information from the Java analysis with that contained in the Oracle documentation. This integration, as illustrated in Figure 4.2, enabled us to gather a comprehensive set of Java built-in methods.
Joining the data presented its own set of challenges. Our initial approach aimed to merge the extensive dataset of methods from the projects, which was roughly 32GB, with the more concise dataset from Oracle's documentation, which was about 8 MB, depending on the version. However, we encountered memory and computational limitations, making this direct method impractical.
In our search for an efficient solution, we tried to use a MySQL database [21]. We created tables for both datasets and employed indexes on them, aiming to accelerate the joining operation. Indexing can significantly speed up data retrieval times as it allows the database system to fetch the data directly without scanning every row in a table, which could make a big difference when dealing with such large datasets [22]. However, the amount of data combined with its complexity of the join operation caused system crashes. Although we used these indexes and experimented with different optimization techniques, the challenges remained and we could not resolve the system stability issues.
Given the challenges encountered with the MySQL database, we decided to use Python's pandas library [23]. The pandas library is a well-known for its data manipulation capability, especially when dealing with large datasets. It offers a flexible and efficient DataFrame structure[24] that can support large amounts of data, and its in-memory operations can be significantly faster than traditional database joins, especially when appropriate optimizations are applied.
Direct merging, as we did with the tables in MySql, took too much time and consumed a lot of resources. Due to the risk of system crashes, we halted the operation after approximately 5 hours and shifted to a segmented approach. We partitioned the 32GB dataset into smaller chunks, each approximately 1GB in size. Working with these manageable chunks allowed for more efficient

[21] MySQL https://www.mysql.com/
[22] Use of Index in SQL Server https://learn.microsoft.com/en-us/sql/relational-databases/sql-server-index-design-guide?view=sql-server-ver16
[23] pandas https://pandas.pydata.org/
[24] DataFrame https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.html

operations, and when we individually joined each with the Oracle data, the results were promising. The join process took between 20 and 50 minutes per chunk, depending on the chunk. However, this segmented joining process led to a substantial increase in the overall volume of data, resulting in an aggregated dataset size of around 1.4TB after processing all chunks.
Such a large dataset comprised of the Java methods we extracted and the Oracle documentation for a particular version that we scraped. For this reason this large dataset necessitated further refinement. We broke down each large merged file into smaller segments for more effective filtering. We then applied three different filters, focusing on various method and class details such as parameters, parameter types, and associated libraries. Following we have a brief description of each filter:

1. **Method Name:** With this first filtering process, we removed all the rows that had different names in the Java class and in the Oracle documentation. This resulted in a new dataset where we had only rows with the same names.
2. **Parameter Consistency**: From the Java analysis, we were also able to retrieve the number and type of parameters that a method requires. A similar extraction was done from the Oracle documentation. With these two pieces of information, we can further refine the rows and select only those that have the same number of parameters and type.
3. **Presence of Library:** In most cases one need to import the corresponding library to use a method. The same is true for Java methods. From the Oracle documentation we knew to which class a method belonged, and from the Java analysis we had a list of imported libraries. During the last filtering process, we checked whether the required library was included in the list of imported libraries of the respective row.

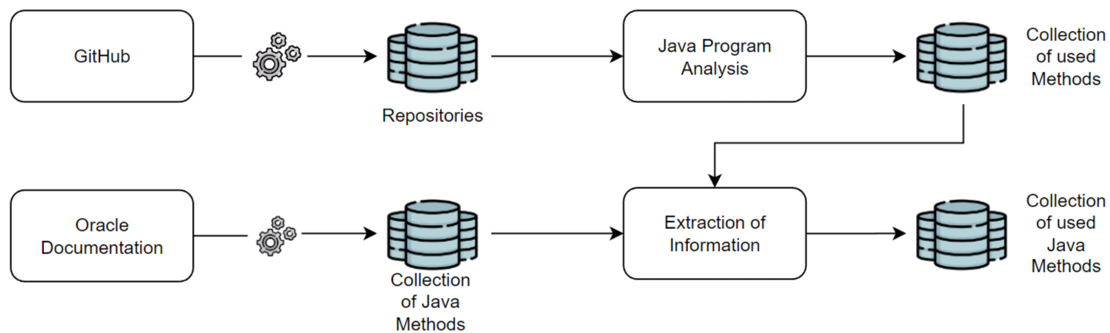

Figure 4.2: Process Overview: Collecting Data on Java Built-in Methods Usage

It is also important to note that we repeated the process of partitioning, merging, and filtering for each LTS Java version, namely Java 8, 11, and 17. The mapping process took the most time compared to the previous implementation and analysis.

4.6 Survey

Our sample is from open-source projects hosted on GitHub, so the data is representative of that specific domain. However, we also wanted to understand the perspective of industry professionals where coding standards might differ from those in open-source projects on GitHub. To address this gap, we decided to conduct a survey with closed-ended questions. We opted for closed-ended

questions to obtain quantifiable data, facilitating objective analysis and pattern recognition [Cln13]. In addition, empirical studies, such as those by Zhou et al. [ZWZG17], have shown that choosing closed-ended questions can lead to a more robust response rate from participants. In this particular study, over 75.0% of respondents did not answer open-ended questions when given the choice. This significantly high non-response rate highlights the potential challenges that open-ended questions present in certain contexts. Therefore, to ensure consistent participation, we chose closed-ended questions. These questions also simplify the data collection process, making it efficient for both participants and researchers. With the data available to us, we were able to formulate the questions in a way that only allowed for certain choices related to the results obtained. In this way, we were able to minimize missing data, which could be a problem with open-ended questions.
Since our survey involved human subjects, we adhered to the four basic principles defined by Singer and Vinson in 2002 [SV02], namely informed consent, scientific value, beneficence, and confidentiality. Adherence to these four principles ensures that our survey does not harm participants in any way and that the objectives and data collection methods are clearly understood and consistent with the research scope. All information on the four principles comes from Singer et al. [SV02]. Following is a brief description of how we implemented these principles:

1. **Informed Consent:** With this principle, we must ensure that each participant knows about the purpose, methods, potential risks, and benefits of the research before giving consent to participate. They should give their consent voluntarily and without coercion. We published the invitation to the survey together with detailed information about the objectives, procedures, and benefits of the study. This ensured that potential participants were fully informed before they decided to take part in the survey.

2. **Scientific Value:** This principle supports the importance of ensuring that research contributes to meaningful knowledge or advances knowledge in its respective field [Mil96]. Although some surveys examined the distribution of Java versions, as we saw in the introduction, and the use of some libraries, they did not examine the use and adoption of specific Java built-in methods and their impact on software quality. In this way, we ensure that our survey addresses gaps in current knowledge and has the potential to provide significant insight into our research topic.

3. **Beneficence:** The beneficence principle aims to maximize potential benefits while minimizing potential risks or harms to participants. To achieve this, we first tested our survey with seven participants who gave us their feedback. This way, we can ensure that there are no annoying or unclear questions in the actual survey. The feedback from the pilot helped us refine the survey to ensure participant comfort and to collect high-quality data.

4. **Confidentiality:** The last principle is based on protecting the identity of the participants. The identity and responses of participants should remain private and protected to ensure that no personal information can be traced back to an individual. All survey responses were collected anonymously, ensuring that no data could be traced back to any individual. In addition, the survey was conducted in a group with many participants, making it difficult to trace back to an individual participant. The survey results were stored securely with restricted access.

In preparing the survey questions, we ensured direct correlation to our primary research questions. This was done to provide a robust discussion of the results from open-source projects and to allow for comparative analysis with survey results.

The design of our survey was influenced by established research methods. In particular, we consulted previous research that either conducted similar surveys or described relevant techniques for survey design in academic research [Sch11][DSG+22]. Our overall goal was to maximize respondent participation. To achieve this, we emphasized clarity, brevity, and structured organization in the survey design.
A first version of the survey was tested with a group of seven participants. Three of them had 1-3 years of Java experience, while the rest had more than 5 years. Their participation provided valuable feedback on the survey's comprehensibility, clarity, structure, and objectives.
This first iteration was invaluable in gaining feedback. In particular, the participants pointed out ambiguities in questions about the frequency of method usage within different Java built-in methods. Questions on code smells and bugs were also criticized for their broad scope. Participants differed in their interpretation of code smells and bugs, highlighting a lack of consensus on their definitions. To address these concerns, we revised the questions to improve their clarity and ensure more consistent interpretation, and thereby increase the reliability of the data collected.
After the feedback and improvement of some questions, we came up with a total of 16 questions, of which the first four are mandatory and the rest are optional. The questions for the survey can be seen in Table 4.7. We first starting with the demographic categorization based on experience of the candidates. We then proceed into questions related to our research, with which we can compare the results from the samples and the answers we get.
Following the feedback from the first iteration of the survey, we changed the question type regarding the SonarQube part and introduced a description between question 10 and 11 indicating the option to skip and conclude the survey if no knowledge or use of SonarQube was provided by the participant. We also changed the question type regarding SonarQube from a scale question to a Yes/No question. Meanwhile, the description with some examples was removed from the questions about the use of the Java built-in methods (5-10), as this caused influence on the choice and confusion among some participants. Reading the additional description and example for each question also increased the time it took each participant to complete the survey.

Nr.	Question	Type	Scope
1	How many years have you been actively programming in Java?	Multi-Choice (Single Select)	Demographic
2	Which version of Java do you use the most in your projects?	Multi-Choice (Single Select)	Java Project Version
3	How often do you rely on Java built-in methods?	Slider	Adoption Ratio
4	In which domain do you work the most?	Multi-Choice (Multiple Select)	Domain Distribution
5	Out of these built-in methods from Java 8, which do you use the most?	Multi-Choice (Single Select)	Adoption Ratio
6	Out of these built-in methods from Java 8, which do you use the least?	Multi-Choice (Single Select)	Adoption Ratio
7	Out of these built-in methods from Java 11, which do you use the most?	Multi-Choice (Single Select)	Adoption Ratio
8	Out of these built-in methods from Java 11, which do you use the least?	Multi-Choice (Single Select)	Adoption Ratio
9	Out of these built-in methods from Java 17, which do you use the most?	Multi-Choice (Single Select)	Adoption Ratio
10	Out of these built-in methods from Java 17, which do you use the least?	Multi-Choice (Single Select)	Adoption Ratio
11	Upon adopting Java version 8 in your projects, did you observe an increase in Code Smells identified by SonarQube compared to previous versions of Java?	Binary Yes/No	SonarQube Code Smells
12	Upon adopting Java version 11 in your projects, did you observe an increase in Code Smells identified by SonarQube compared to previous versions of Java?	Binary Yes/No	SonarQube Code Smells
13	Upon adopting Java version 17 in your projects, did you observe an increase in Code Smells identified by SonarQube compared to previous versions of Java?	Binary Yes/No	SonarQube Code Smells
14	Upon adopting Java version 8 in your projects, did you observe an increase in Bugs identified by SonarQube compared to previous versions of Java?	Binary Yes/No	SonarQube Bugs
15	Upon adopting Java version 11 in your projects, did you observe an increase in Bugs identified by SonarQube compared to previous versions of Java?	Binary Yes/No	SonarQube Bugs
16	Upon adopting Java version 17 in your projects, did you observe an increase in Bugs identified by SonarQube compared to previous versions of Java?	Binary Yes/No	SonarQube Bugs

Table 4.7: Overview of the Questions from the Survey

4.7 Metrics

Metrics provide a structured approach to understanding the vast world of software development. They can convert elusive concepts into concrete numbers and provide insight into a sea of ambiguity [RMN19]. Such metrics are critical when examining trends in software adoption. They allow us to identify both the occurrences and the underlying motivations. Our research introduces a set of metrics focused on assessing the quality and integrity of Java built-in methods and code. According to Kan in his book "Metrics and Models in Software Quality Engineering" [Kan14], software metrics can be roughly divided into three categories: Product Metrics, Process Metrics, and Project Metrics. Product metrics capture the characteristics of the software product and deal with its size, complexity, and overall quality level. We decided to use metrics such as "Java built-in methods per KLOC", "Code Smells per KLOC" and "Bugs per KLOC" for our research, since our research is mainly focused on the category of product quality metrics. This normalization technique allows us to make fair comparisons and assessments across projects, regardless of their size.

5 Results

While our methodology is based on a large dataset containing a vast amount of information, it is largely exploratory and has quantitative elements from the survey. We did not approach the data with predefined hypotheses, but were guided by our research question. Our goal was to examine the data to uncover underlying patterns and correlations in order to formulate statements for future research. In this chapter, we present the results of our project analysis, which are guided by the research questions.

5.1 Adoption of Java Version Features (RQ1)

In the following sections, we will provide a clearer overview and deeper understanding of how the adoption of new Java version features is manifested in projects on GitHub.

Degree of New Java Feature Adoption (RQ1.1)

Table 5.1 provides an overview of the use of the Java built-in methods in open-source Java projects. In this table, we can see that 151,185,183 LOC were analyzed. When looking at the methods used in this codebase, a total of 85,887,864 methods were found. Interestingly, when excluding setters and getters (which are commonly used for encapsulation and can inflate the method count), the number of methods decreases to 58,887,603.
If we focus only on the Java built-in methods, the data shows that 155,545 methods from this large codebase are directly linked to Java built-in methods. To put these numbers into context in terms of code size, the data provides further insight into method density. For example, on average, there are 568.21 methods per KLOC. However, when we filter out the setters and getters, this density still aligns closely with the previous value, 559.5 methods per KLOC.
With respect to the key question about the adoption rate of methods of the Java version, it is interesting to note that the Java built-in methods have a density of only 1.03 per KLOC. Furthermore, if we consider their percentage of the total number of methods, the Java built-in methods account for 0.18% of all methods. This percentage increases slightly to 0.26% when considering the total number of methods without setters and getters, having 1.478 Java built-in methods per KLOC.

Figure 5.1 shows the distribution of method occurrences over a number of Java versions.
At the top of this distribution is Java 6 with a high value of 59,121 method occurrences. Three other versions that also have a large number of built-in methods are Java 1.0, 1.2, and Java 1.4, which, with 39,700, 28,855, and 18,863 method occurrences, respectively.
At the other end of the spectrum, the newer versions from Java 8 through 16 are sparsely represented

Metric	Value
Total LOC	151,185,183
Total LOC Without Set/Get	105.285.183
Total Numbers of Methods	85,887,864
Total Numbers of Methods Without Set/Get	58,887,603
Total Java Built-in Methods	155,545
Java Built-in Methods per KLOC	1.03
Java Built-in Methods per KLOC Without Set/Get	1.478

Table 5.1: Java Built-in Methods Results Overview

with method counts ranging from 1 to 228. In contrast, the older versions of Java, such as 1.1, 1.3, 5, and 7, show a more irregular pattern. While some, such as 1.1 and 7, anchor themselves in the 1,000 to 7,000 range, the others remain below 1,000.

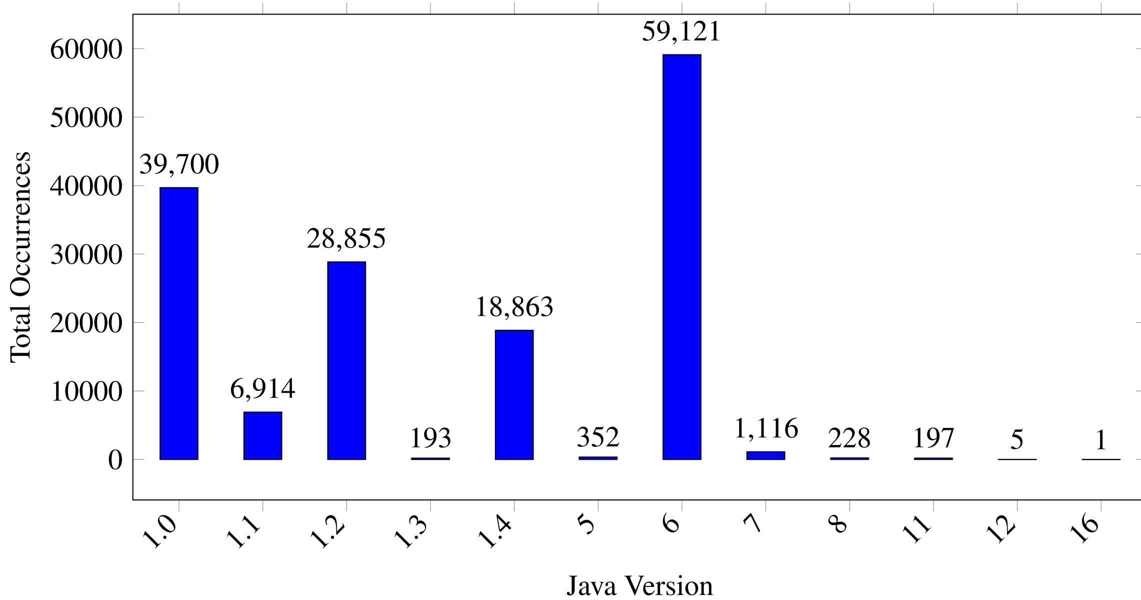

Figure 5.1: Distribution of Java Versions Weighted by Method Occurences

Figure 5.2 visually shows the sources of our data on the use of Java's built-in methods in different projects. The "radiance" project is the main contributor with 37.55% of the data on Java method usage. It is followed by "teamv" and "bc-java", which contribute 6.6% and 1.98%, respectively. These three projects are the main sources of our data on the use of Java's built-in methods.
The "Others" category contributes 19.29% to our data pool. This category includes projects that each provided less than 1.0% of the data on the use of Java's built-in methods.
The segment labeled "NO_DATA" accounts for 34.67% of our data sources. This percentage represents the projects from which we were unable to extract data on Java method usage due to the

limitations of our analysis method. Essentially, we are missing data from 750 projects on the use of newer Java built-in methods.

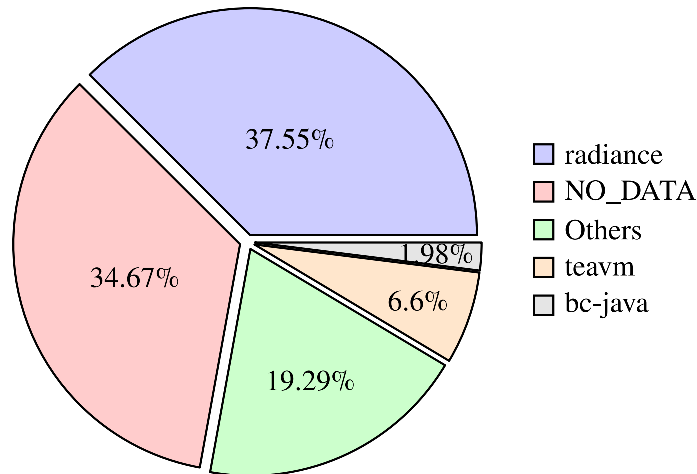

Figure 5.2: Distribution of Java Built-in Methods Usage Across Analyzed Projects

Most Adopted Java Methods on GitHub (RQ1.2)

Table 5.2 provides a detailed breakdown of the three most commonly used methods for each Java version.
For Java 1.0, the analysis shows a strong bias towards date and time functionality. The "time.LocalDate:of" method stands out with a remarkable 6,101 usages. It is closely followed by "util.Properties:getProperty" with 3,007 usages and "time.LocalTime:of" with 2,654 usages.
The trend shifts slightly with Java 1.1, where graphical and user interface components take a more central role. Here, the "awt.Component:setBounds" method is especially preferred with 2,174 usages. Interesting is the parallel preference for "awt.Rectangle:setBounds", which received 2,151 usages. Also, "util.Calendar:set" shows a continuation of the popularity of time-related methods with 1,023 usages.
The "util.Properties:setProperty" method dominates the chart with a whopping 13,304 usages, indicating its widespread use in property configurations. In addition, file operations seem to have gained attention, as shown by "io.File:createTempFile" with 4,658 usages.
The "util.Timer:schedule" method was used particularly frequently, with 61 usages. Meanwhile, reflection-based operations gained traction, as evidenced by the nearly equal usages of "reflect.InvocationHandler:invoke" and "net.URLStreamHandler:equals", both rounding up to 40 usages.
In Java 1.4, the preference patterns shifted to the area of preferences and configuration settings. The "prefs.Preferences:put" method tops the list with 7,561 usages. Its counterparts "prefs.AbstractPreferences:put" and "prefs.Preferences:get" are close behind with 7,150 and 906 usages, respectively, underscoring a clear preference for customizing and configuring settings in this version.
Java 5 saw increased usage of methods related to concurrent programming and graphical components. The "io.Wire:append" method was used 48 times, while "Atomic[1]:compareAndSet" (which refers

to the family of atomic operations) and the graphical method "awt.Inset:set" were mentioned 37 and 36 times, respectively.
With the transition to Java 6, geometric and window-based operations were used. The "geom.Path2D.Float:curveTo" method topped the hit list with an impressive 57,404 usages. It was followed by window matching operations such as "awt.Window:setBounds" with 763 usages and the "geom.Path2D.Float:quadTo" method with 126 usages.
Java 7 focused on object manipulations and file operations. The "util.Objects:equals" method was used 764 times. The methods for comparing and creating files, "file.Files:isSameFile" and "file.Files:createLink", were used 72 and 70 times, respectively, underscoring the importance of file operations in this version.
In Java 8, stream-based operations gained importance. The "util.Arrays:stream" method was used 70 times, followed by "stream.IntStream:concat" and "util.Calendar.Builder:setDate" with 67 and 39 usages, respectively, illustrating the growing importance for stream operations and data processing.
When looking at the newer Java 11, builder patterns stand out, with the "Builder[2]:header" method used 140 times, followed by its variants with 37 and 20 usages.
Finally, for Java 12 and 16, the adoption of the new methods in the GitHub repositories is limited, with only 5 and 1 usages, respectively.
To put the results in context, we need to remember that these results come from a pool of 155,545 Java built-in methods. The highlighted methods therefore represent those that have found the most adoption among the developer community on GitHub across the different Java versions.

5.1.1 Least Adopted Java Methods on GitHub (RQ1.3)

As we did before, we also have a table showing the least adopted Java built-in methods. Table 5.3 presents the three least utilized methods across various versions of Java, from version 1.0 to version 16. These results provide insights into the less commonly used methods in each Java version.
For Java versions 1.0 through 8, the least utilized methods primarily emanate from packages like java.sql, java.awt.image, and java.util. The methods from these versions have an extremely low utilization count, mostly registering a single occurrence. This suggests that, within the datasets analyzed, these specific methods were either very specialized or were less known among the developer community.
In the later versions, especially from Java 11 onwards, the least used methods mainly belong to the java.net.http and java.nio packages. Due to the limited number of Java built-in methods identified for these versions, there is an overlap between the most and least used methods.

5.1.2 Evolution of Java Method Adoption (2011-2023) (RQ1.4)

Figure 5.3 shows a chronological visualization of the distribution of Java built-in methods per KLOC over a 13-year period, from 2011 to 2023. The graph begins in 2011 with a value of 0.45 Java built-in methods per KLOC.
There is a significant decrease from 0.45 in 2011 to 0.17 in 2012. In 2013, there is a slight increase to 0.26, which represents a slight upswing after the decrease in the previous year.
In the following years, 2014 and 2015, there is a further decrease, with values of 0.23 and 0.17,

Version	1st Most Used	2nd Most Used	3rd Most Used
1.0	time.LocalDate:of (6101)	util.Properties:getProperty (3007)	time.LocalTime:of (2564)
1.1	awt.Component:setBounds (2174)	awt.Rectangle:setBounds (2151)	util.Calendar:set (1023)
1.2	util.Properties:setProperty (13304)	io.File:createTempFile (4658)	security.KeyFactory:getInstance (1968)
1.3	util.Timer:schedule (61)	reflect.InvocationHandler:invoke (40)	net.URLStreamHandler:equals (40)
1.4	prefs.Preferences:put (7561)	prefs.AbstractPreferences:put (7150)	prefs.Preferences:get (906)
5	io.Writer:append (48)	Atomic[1]:compareAndSet (37)	awt.Insets:set (36)
6	geom.Path2D.Float:curveTo (57404)	awt.Window:setBounds (763)	geom.Path2D.Float:quadTo (126)
7	util.Objects:equals (764)	file.Files:isSameFile (72)	file.Files:createLink (70)
8	util.Arrays:stream (70)	stream.IntStream:concat (67)	util.Calendar.Builder:setDate (39)
11	Builder[2]:header (140)	Builder[2]:setHeader (37)	Builder[3]:header (20)
12	file.Files:mismatch (5)	-	-
16	nio.ByteBuffer:put (1)	-	-

-Atomic[1]: atomic.AtomicIntegerArray - Builder[2]: http.HttpRequest.Builder - Builder[3]: http.WebSocket.Builder

Table 5.2: Top 3 Most Used Methods by Version

respectively. A significant increase is observed between 2015 and 2016, where the value increased from 0.17 to 0.59, and there is also a significant jump between 2021 and 2022, from 0.61 to 0.91. After 2015, the trend gradually goes up year by year. In 2023, the use of Java built-in methods per KLOC reaches a peak of 1.03, indicating a consistent increase in their use.

For a closer look, we can see the evolution of Java built-in methods by version over the years in Figure 5.4 and Figure 5.5, where we can see the changes and adoption through the years of Java 8, 11, and 17. Both diagrams offer a visual representation of the distribution of Java built-in methods from 2011 to 2023. Due to the small number of methods in versions 11 and 17, we analyzed them separately from version 8 for a clearer overview.
Starting with the first diagram, we can see that there is a consistent trend in methods for versions 1.0 and 1.2 between 2011 and 2015, with version 1.0 peaking in 2013 with 2159 methods. Interestingly, version 1.4 begins to make its presence felt in 2014. In the period from 2016 to 2018, version 1.4

Version	1st Least Used	2nd Least Used	3rd Least Used
1.0	sql.PS:setObject (1)	AWT[1]AASF:setPixels (1)	AWT[1]CIF:setPixels (1)
1.1	util.STZ:setStartRule (1)	util.STZ:setEndRule (1)	awt.AEM:add (2)
1.2	sec.Prov:put (1)	AWT[2]AT:setToRotation (1)	AWT[2]RR2D.D:setRoundRect (1)
1.3	IMD[5]:getIMDisplayName (1)	net.USH:hostsEqual (2)	net.USH:setURL (3)
1.4	awt.f.GV:getPixelBounds (1)	lang.Str:replaceAll (1)	nio.CB:wrap (1)
5	awt.Comp:firePC (1)	util.ca.a.AMR:compareAndSet (1)	atomic.ASR:compareAndSet (1)
6	awt.f.LP:pathToPoint (1)	net.HC:domainMatches (1)	awt.f.LP:pointToPath (1)
7	util.Cal:setWeekDate (1)	MXBean[3]:setLL (1)	sql.DM:getPseudoColumns (9)
8	util.Cal.B:setWeekDate (1)	util.S:spliterator (1)	util.A:spliterator (2)
11	http[4].HR.B:header (20)	http[4].HR.B:setHeader (37)	http[4].WS.B:header (140)
12	nio.f.F:mismatch (5)	-	-
16	nio.BB:put (1)	-	-

AWT[1]: awt.image. AWT[2]: awt.geom. MXBean[3]: lang.management.PlatformLoggingMXBean. http[4]: net.http. IMD[5]: awt.im.spi.IMD.

Table 5.3: Top 3 Least Used Methods by Version

records a sharp increase from 7018 methods to a remarkable 11219 methods in 2018, highlighting its great influence in these years.
However, the year 2022 is characterized by a large upswing in the number of methods for different versions. In particular, version 1.0 increases to 33,267 methods and version 6 increases to 57,761 methods, representing a significant shift in method adoption for these versions. The landscape does not appear to change until 2023. The number of methods for versions 1.0 and 6 is 38,595 and 58,961, respectively, and version 1.2 is also clearly on the rise with 26,941 methods.
It is noticeable that versions 5, 1.3, and 8 show a lower number of methods through the years, but version 8 appears in the data from 2016 and shows a steady increase, starting with only 16 total methods in 2016 to 228 in 2023. In addition, certain versions such as 7 and 1.3 show a decreasing trend.

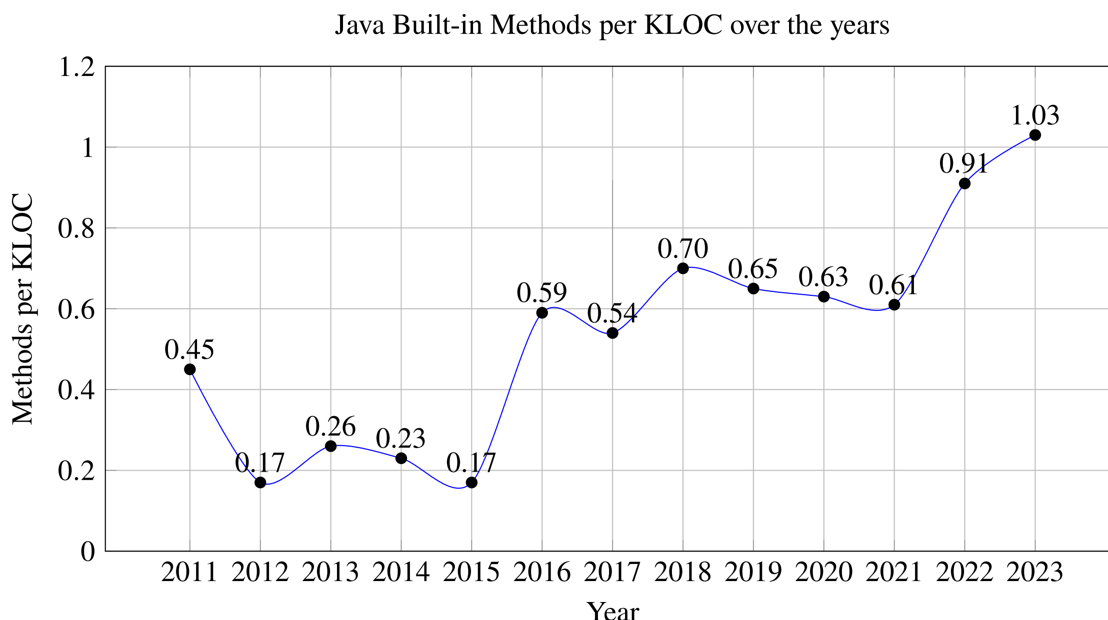

Figure 5.3: Distribution of the Java Built-in Methods per Version over the years

Proceeding with the diagram 5.5, we can see that version 11 had a significant increase to 100 methods in 2020. In 2021, there was a slight decrease to 89 methods. However, in 2022 there was another substantial jump, this time to 165 methods. By 2023, it reached 197 methods.
Version 12 did not have any built-in methods until 2023, at which point it introduced 6 methods. The same is the case for version 16 as for version 12: no built-in methods were found from 2019 to 2022, and in 2023 there is only one method belonging to version 16.

5.2 Java Feature Adoption and Software Quality (RQ2)

Table 5.4 shows data on software quality metrics related to code smells and bugs. We found a total of 380 identified bugs related to specific Java versions. After normalizing this number relative to the total size of the analyzed codebase, we found a value of 0.0025 bugs per KLOC. There are also 4,900 code smells related to a specific Java version. After normalization, the result is a value of 0.0324 code smells per KLOC.

In Figure 5.6, we look at the distribution of code smells and bugs for each Java version per rule type in SonarQube. From the data, it is evident that the main versions with identified bugs are Java 1.1, 1.2, and 8, with counts of 91, 92, and 78, respectively. Interestingly, other versions such as 1.4, 5, 6, 7, and 9 have no or only minor bugs.
Java 8 stands out with a significant number of 1,899 code smells followed by Java 5 and 9 with 752 and 623, respectively. Java 1.2, 1.4, 6 and 7 also have significant numbers with 688, 203, 197 and 594 code smells, respectively. Java 1.1 has the fewest code smells with only 24 instances.

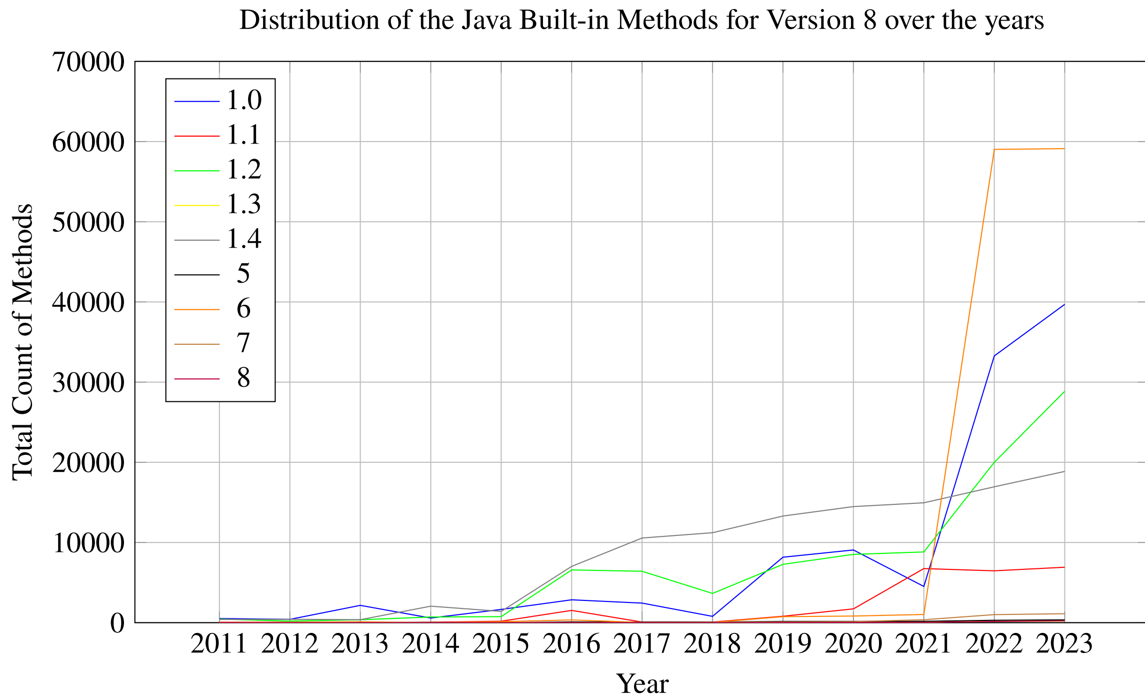

Figure 5.4: Distribution of the Java Built-in Methods for Version 8 over the years

Metric	Count/Value
Number of Bugs	380
Bugs per KLOC	0.0025
Number of Code Smells	4,900
Code smells per KLOC	0.0324
Total LOC	151,185,183

Table 5.4: Number of Bugs, Code Smells, and their Normalized Metrics

We can also see the distribution of priority into which each rule is divided. Figure 5.7 shows the distribution of priority levels for each Java version per rule type in SonarQube.
The "CRITICAL" Priority category draws attention primarily to Java 1.4, with 203 instances. There is also a relatively high number for Java 6 and 7, with 197 and 198 instances, respectively. Java 5 and Java 1.1 have fewer instances, with 36 and 15 instances, respectively. It is noticeable that Java 1.2, 8, and 9 completely lack critical problems.
In the "MAJOR" priority category, Java 8 has the highest number of instances with 1,184. Java 5 also registers a high number with 652 instances. At the bottom is Java 1.2 with 248 instances. While Java 1.4 stands out in the "CRITICAL" category, its absence in the "MAJOR" Priority is quite a contrast.

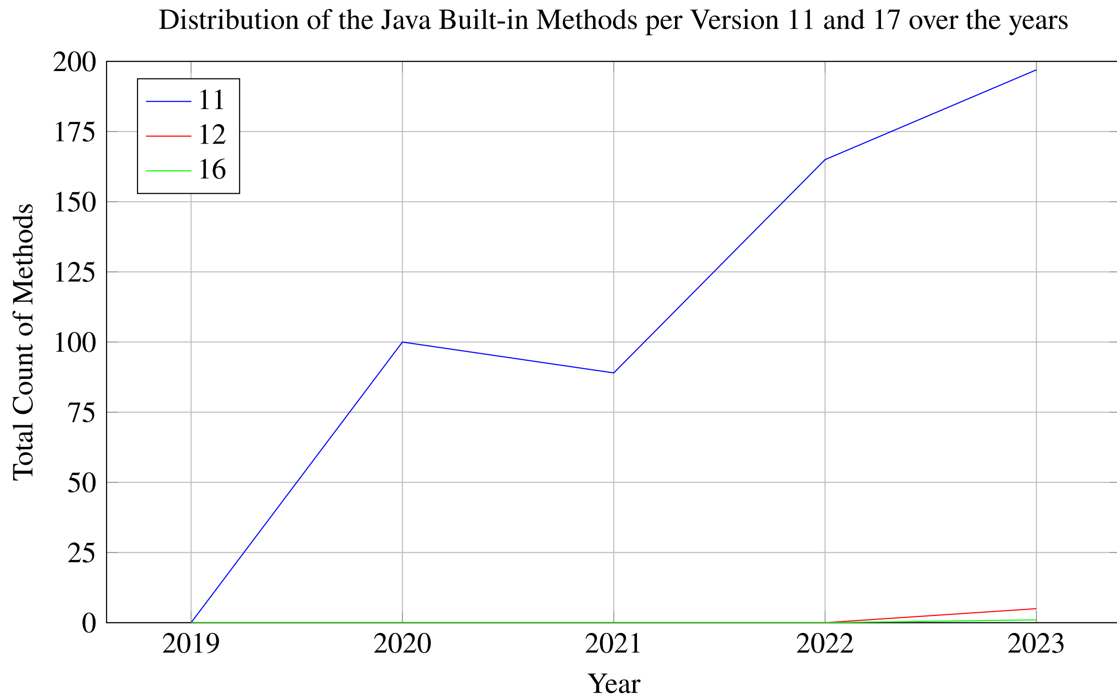

Figure 5.5: Distribution of the Java Built-in Methods per Version 11 and 17 over the years

The “MINOR” priority is dominated by Java 8 and Java 9, with 793 and 612 instances, respectively. Java 1.2 is still very much in evidence, with 532 instances.

5.3 Factors Influencing Java Feature Adoption: Domain, Contributors, and Project Size (RQ3)

Figure 5.8 shows a comprehensive visual representation of the distribution of Java versions across different domains, weighted by the percentage occurrence of integrated Java built-in methods.
Starting with the domain “Application Software”, a clear preference for Java 1.2 and 1.4 can be seen with a share of 46.0% and 33.0% of the total Java built-in methods for this domain, respectively. Java 1.1 is also very strongly represented with a share of 12.5%. Later versions from Java 5 onwards have a drastically lower distribution, with the percentage rarely exceeding 1.0%. The notable exceptions are Java 6 and 7, which have 3.2% and 2.8%, respectively.
In the “Documentation” domain, the Java 1.2 version dominates with 51.1% of the total Java built-in methods for this domain. This domain also shows a strong inclination towards Java 1.4, which accounts for 27.0%. The presence of Java 7 is considerable at 6.3% and exceeds its share in the “Application Software” domain.
The third domain, “Non-Web Libraries and Frameworks”, is also dominated by Java 1.2 at 48.8% of the total Java built-in methods for this domain, followed closely by Java 1.4 at 28.8%. Java 1.1 and 6 are also well represented with 10.8% and 5.2% respectively.
One domain where Java 1.1 peaks at 15.0% is the “Software Tools” domain. However, this peak

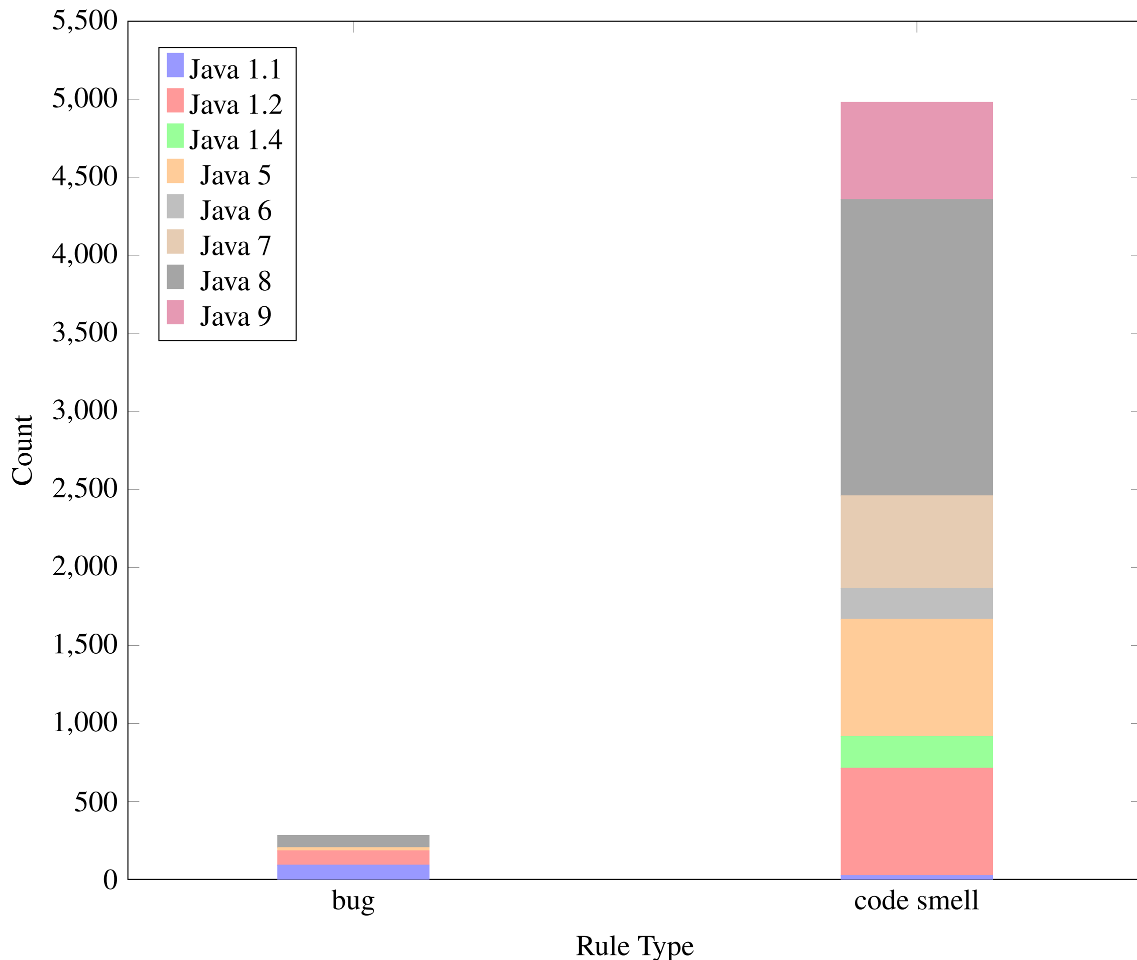

Figure 5.6: Distribution of Code Smells and Bugs for the individual Java Versions per Rule Type in SonarQube

value is still surpassed by Java 1.2 and 1.4, which reach 44.0% and 28.9%, respectively. Java 6 is also strongly represented with 5.5%.
The domain in which Java 1.2 is most frequently used is "System Software" with 54.3% of the total Java built-in methods for this domain. Java 1.4 follows with 29.8%, while the other versions are only minimally represented.
Finally, in the "Web Libraries and Frameworks" domain, Java 6 is strongly represented with 20.0% of the total Java built-in methods for this domain, only surpassed by Java 1.2 with 39.7%. Java 1.4 holds its own with 23.9%. Interestingly, this is the only domain where Java 6 outperforms Java 1.4. In several domains, Java versions such as 1.3, 8, 11, 12 and 16 often have adoption rates of less than 1.0%. Versions 12 and 16 in particular are often not present at all and reach 0.0% in domains such as "Documentation", "Software Tools", "System Software", and "Web Libraries and Frameworks".

A second factor we considered was the number of contributors each project had. To ensure a fair distribution of contributors across projects, we decided to split the projects based on percentiles of contributor counts. This resulted in four distinct ranges: the 0-25th, 25-50th, 50-75th, and

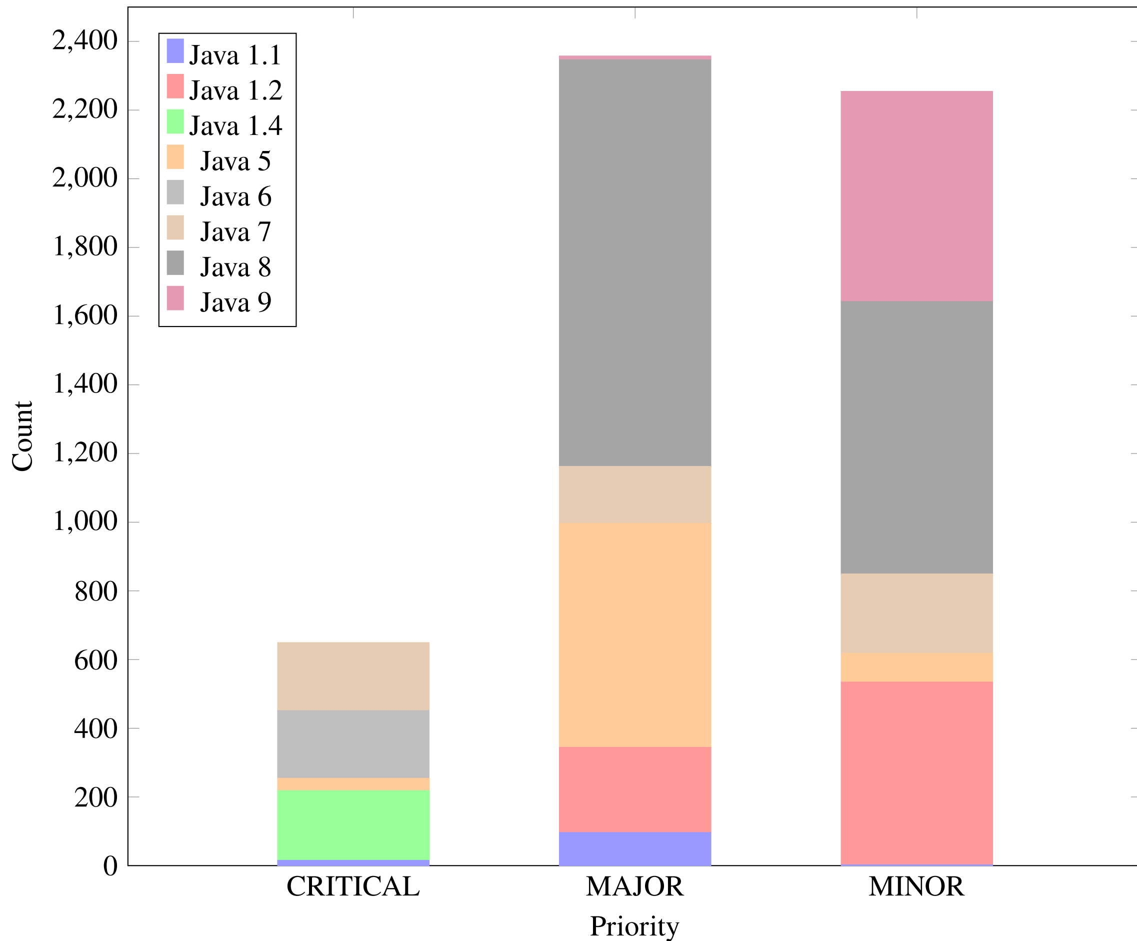

Figure 5.7: Distribution of Priority for the individual Java Versions per Rule Type in SonarQube

75-100th percentiles. For context, projects within the 0-25th percentile had between 1 to 10 contributors. Those falling within the 25-50th percentile had between 11 to 26 contributors. The 50-75th percentile encompassed projects with 27 to 60 contributors, and projects in the 75-100th percentile had a broader range of 61 to 463 contributors.

In the 0-25th percentile, as depicted in Figure 5.9, Java 6 predominates, accounting for 57,763 out of 67,184 Java built-in methods. While Java 1.0, 1.1, and 1.4 also exhibit substantial usage with 3,773, 1,697, and 2,819 Java built-in methods, the least adopted versions in this category are 8, 11, 12, with Java 16 having no representation.

The 25-50th percentile (Figure 5.10) is dominated by Java 1.2, representing 4,811 of the 17,950 methods. Noteworthy are also Java 1.0 and 1.4 with 6,752 and 3,431 methods. However, versions 12 and 16 lack representation in this range.

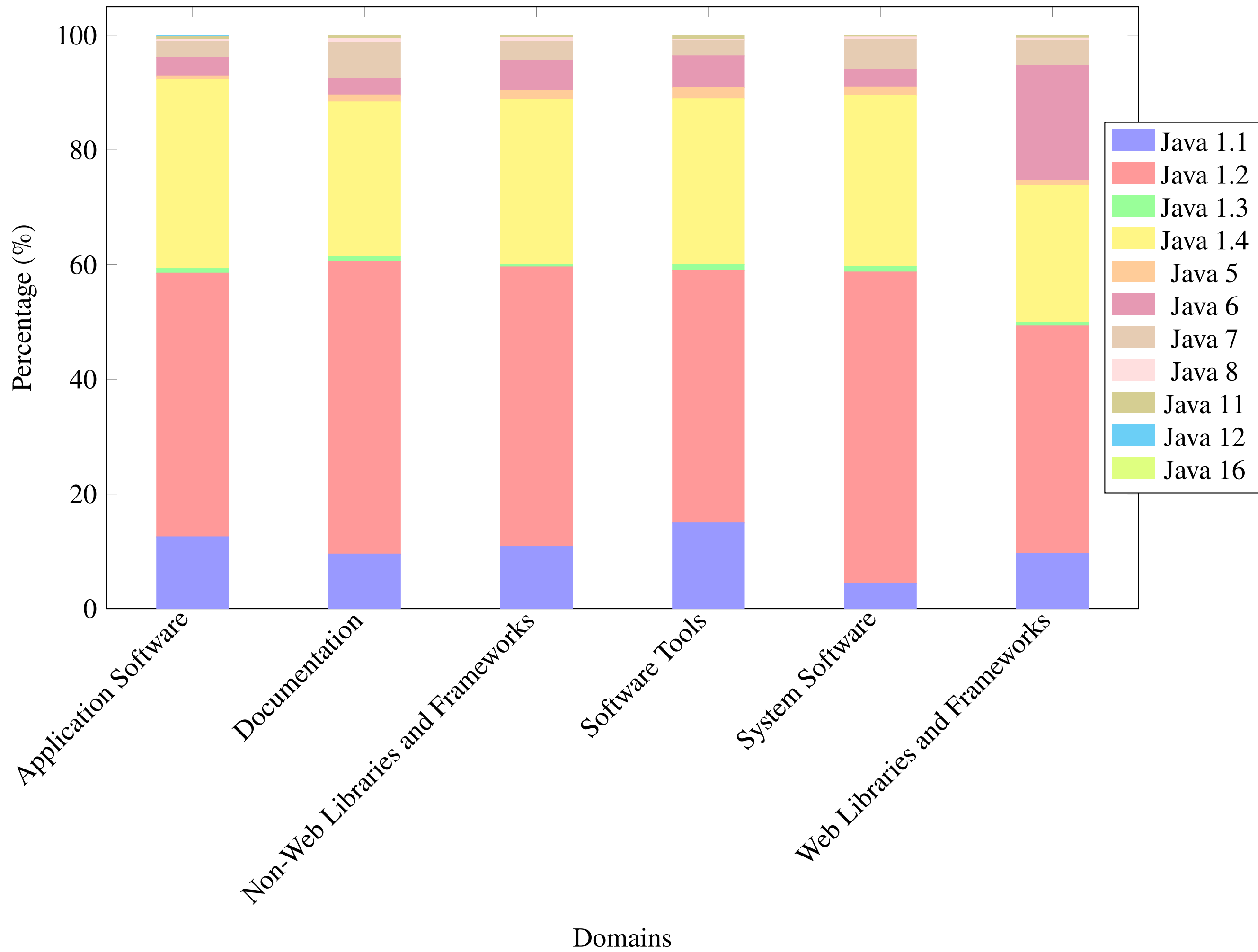

Figure 5.8: Distribution of Java Versions Weighted By Method Occurences in % Across by Domain

Within the 50-75th percentile, illustrated in Figure 5.11, Java 1.0 leads with 11,867 methods. Java 1.2 and 1.4 follow with 4,128 and 3,755 methods, respectively. Conversely, Java 11 and 16 are less prevalent, with 80 and 0 methods, respectively.

Finally, the 75-100th percentile (Figure 5.12) showcases Java 1.2's dominance with 17,968 methods, closely trailed by Java 1.0 with 16,203 methods. A decline is observed with Java 1.4, which possesses 7,327 methods. The remaining versions in this category have lower representations, with Java 12 and 16 being the least adopted, each with just one method.

We also considered the size of the project as factor influencing teh adoption of Java built-in methods, as determined by its lines of code. As we did for the contributors, the data was organized based on percentiles of lines of code of projects. For clarity, the 0-25th percentile contained between 0 and 9,111 LOC. The 25-50th percentile had between 9,111 and 32,488 LOC., the 50th-75th percentile ranged from 32,488 to 108,003 LOC, and the 75th-100th percentile spanned a wider range from 108,003 to 28,063,421 LOC.

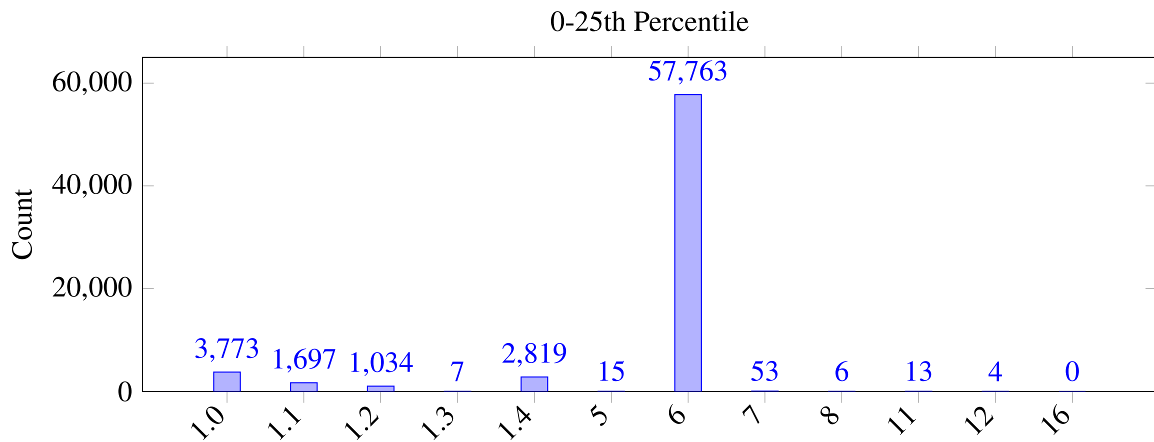

Figure 5.9: Distribution of Methods based on Contributors for 0-25th Percentile

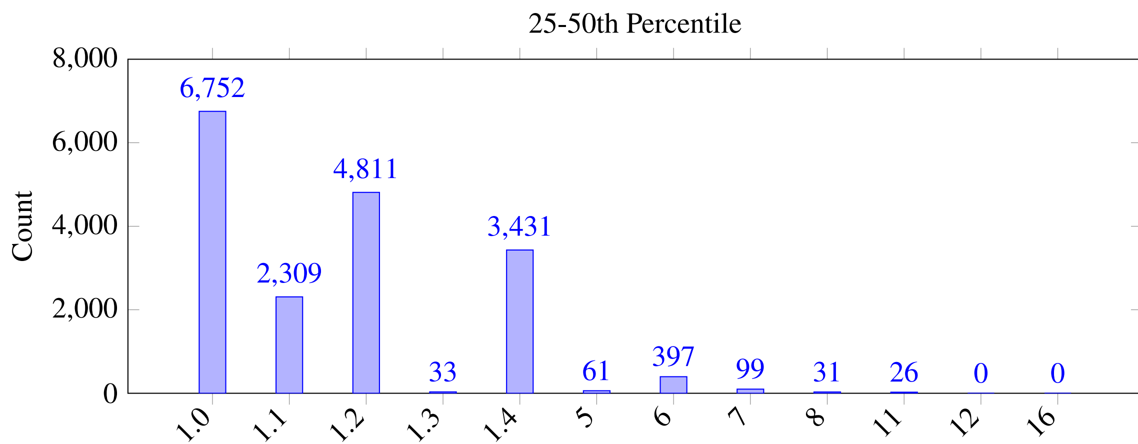

Figure 5.10: Distribution of Methods based on Contributors for 25-50th Percentile

In the 0-25th percentile, as shown in Figure 5.13, Java 1.0 predominates with 441 methods. Java 1.2 and 1.4 also have significant numbers with 359 and 418 methods, respectively. Versions 8, 12 and 16 are not represented in this group, while Java 11 has a minimal presence with 9 methods.

The 25-50th percentile, shown in Figure 5.14, is dominated by Java 1.0 with 2,134 methods. Java 1.4 and 1.2 follow closely behind with 1,705 and 1,172 methods, respectively. Java 12 and 16 remain unrepresented, but Java 11 appears, albeit with a relatively modest count of 19 methods.

Within the 50-75th percentile, as shown in Figure 5.15, Java 1.0 is at the top with 4,944 methods. Java 1.4 and 1.2 remain strongly represented with 3,867 and 2,820 methods, respectively. Java 16 is only minimally represented with only one method, while 12 is not present at all. Meanwhile Java 8 is represented by 36 methods, 10 more than in the previous percentile. Java 11 has a minimal presence, boasting 44 methods.

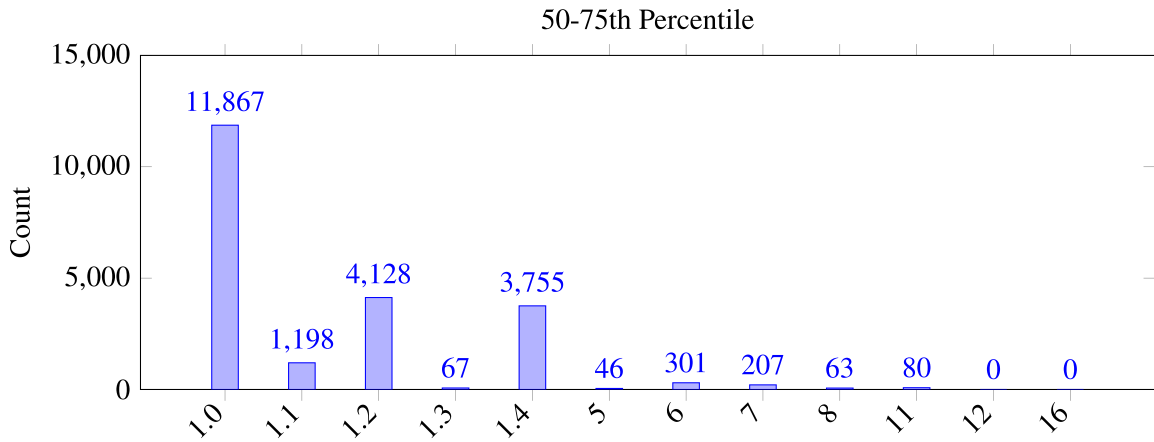

Figure 5.11: Distribution of Methods based on Contributors for 50-75th Percentile

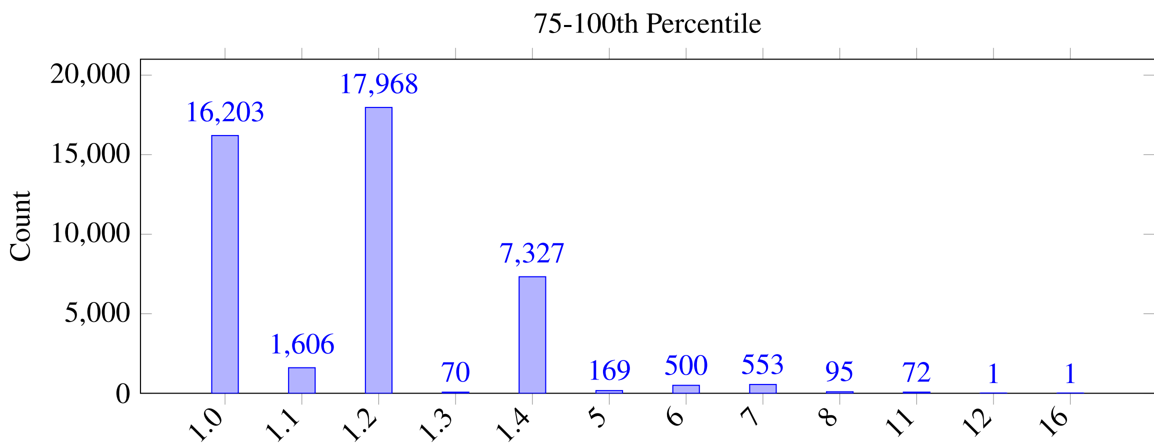

Figure 5.12: Distribution of Methods based on Contributors for 75-100th Percentile

Finally, in the 75-100th percentile, shown in Figure 5.16, Java 6 dominates all others with 58,863 methods. Java 1.0 and 1.2 are also prominent with 32,181 and 24,504 methods, respectively. Java 12 and 16, on the other hand, have minimal to no representation in this group with 5 and 0 methods, respectively. Java 11, in this range, has a modest count of 125 methods.

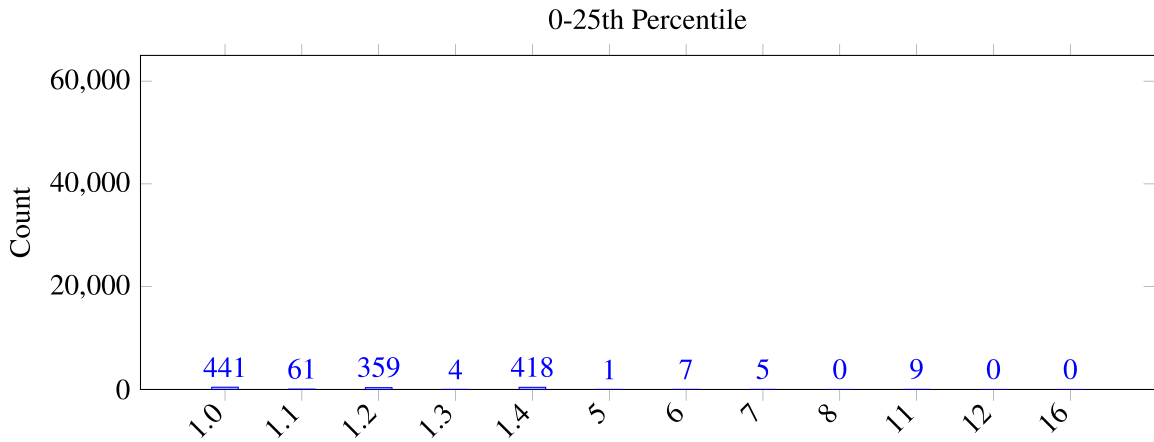

Figure 5.13: Distribution of Methods based on LOC of Projects for 0-25th Percentile

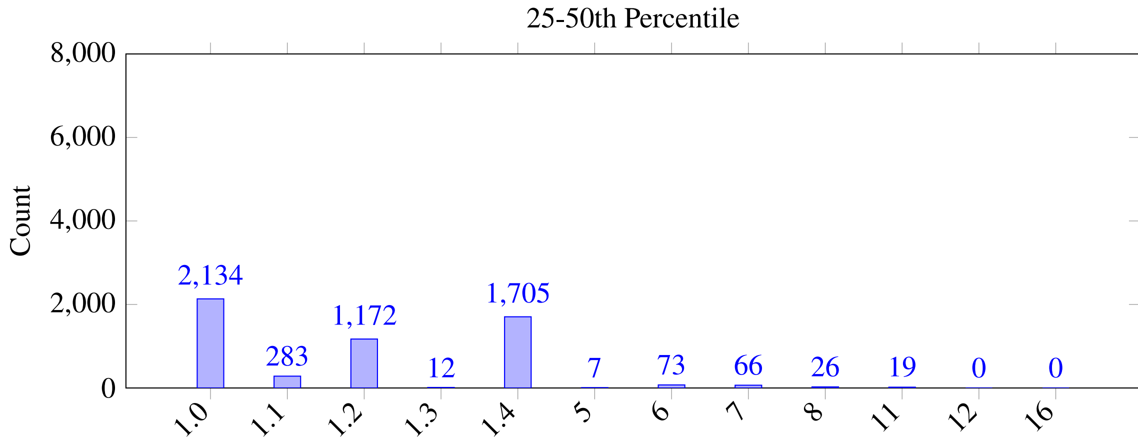

Figure 5.14: Distribution of Methods based on LOC of Projects for 25-50th Percentile

5.4 Java Developers' Experiences vs. Observed Adoption Trends (RQ4)

In this section, we present the survey results corresponding to the questions listed in Table 4.7. A total of 47 respondents with Java experience and some with SonarQube knowledge participated in the survey. It is worth mentioning that the first four questions were mandatory, while the following questions were optional.

For the first question, we have Figure 5.17. This figure illustrates the distribution of the 47 respondents who participated in the survey based on their active years of programming in Java. It is noticeable that the majority, 29 respondents, have extensive experience of more than 5 years.

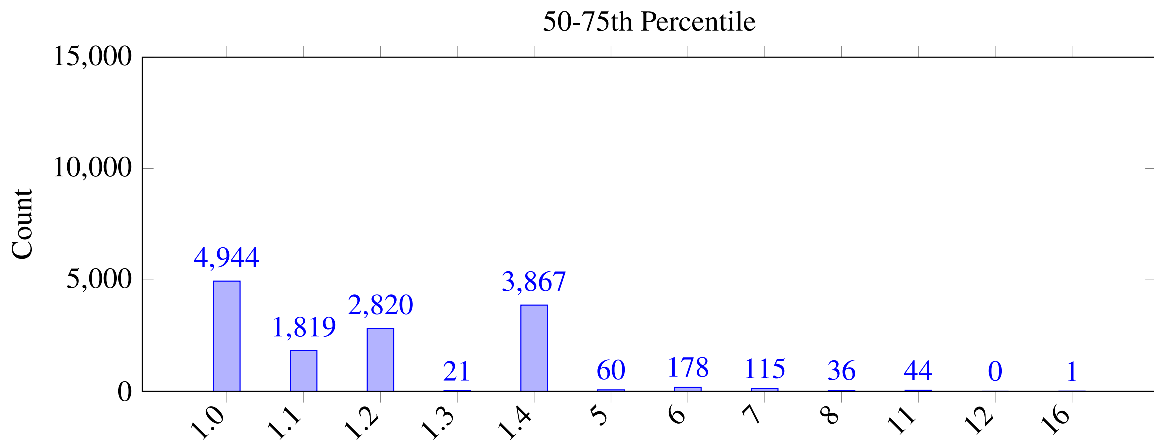

Figure 5.15: Distribution of Methods based on LOC of Projects for 50-75th Percentile

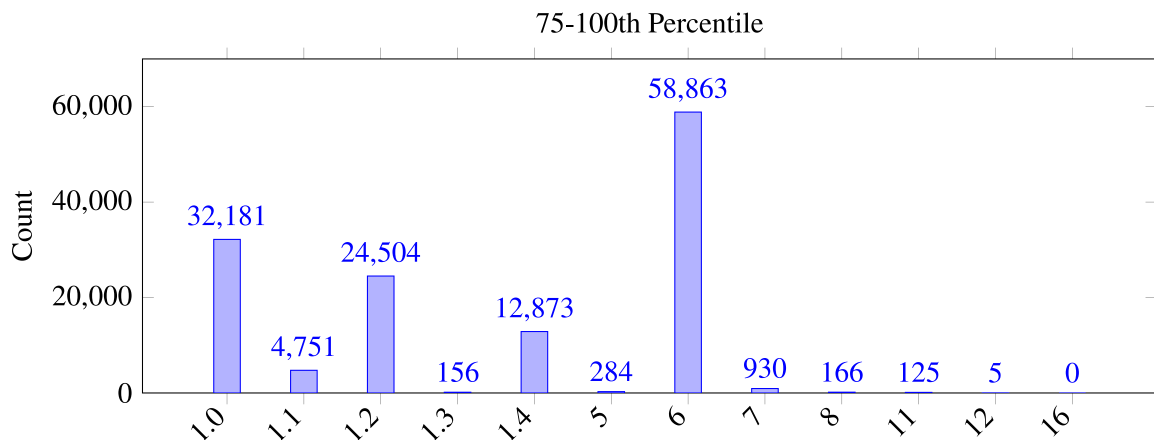

Figure 5.16: Distribution of Methods based on LOC of Projects for 75-100th Percentile

In contrast, only a small subset of 2 respondents have been actively programming in Java for less than a year. In the medium experience range are 10 respondents with 1-3 years of active Java programming, while 6 respondents fall into the 3-5 year range.

For the second question, Figure 5.18 shows the distribution of the Java versions most frequently used in projects among the 47 software developers surveyed. The data shows that Java 8 is the most popular version by a small margin, with 35.0% of respondents using it primarily in their projects. Java 11 and Java 17 are also frequently used, at 33.0% and 30.0%, respectively. A smaller segment, accounting for 2.1%, includes respondents who use other Java versions.

Regarding the reliance on Java built-in methods, we can see it Figure 5.19. This figure shows the reliance of the 47 software developers on Java's built-in methods, as indicated by their rating on a scale from 0 to 10. It is evident that none of the participants rated their reliance from 1 to 3,

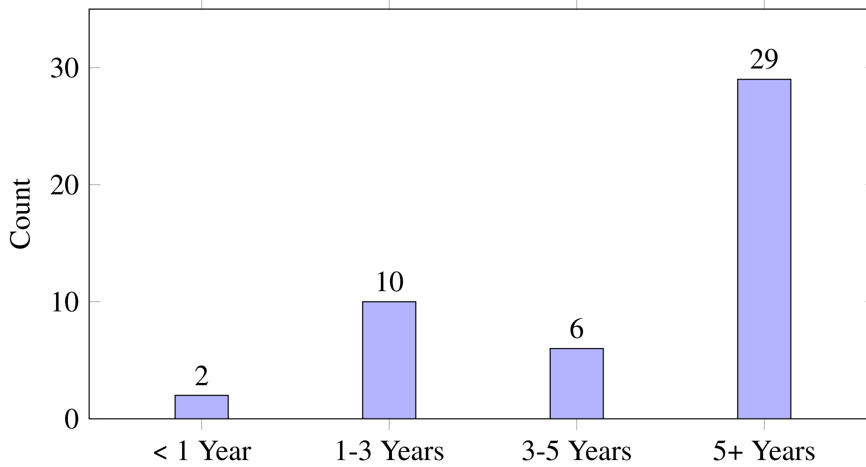

Figure 5.17: Years of Actively Programming in Java by surveyed Software Developers

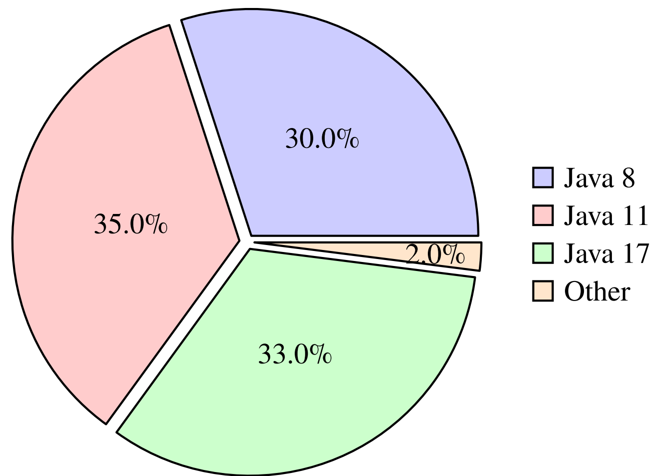

Figure 5.18: Most used Java Version in Projects by surveyed Software Developers

indicating that at least a moderate level of reliance begins with a rating of 4. The ratings of 5, 7, and 10 are particularly notable, with 8, 7, and 9 respondents respectively aligning with these values. The average rating is approximately 7.6. This suggests that, on average, the surveyed respondents have a high reliance on Java's built-in methods

One important question was about the distribution of domains. Here, the respondents could choose between different domains in which they work most. Figure 5.20 illustrates the main domains in which the 47 software developers surveyed work. The vast majority, namely 40 respondents, work mainly in the area of application software. This is followed by "Non Web Libraries/Frameworks" with 8 respondents. "Software Tools" and "Web Libraries/Frameworks" each have 7 and 5 respondents respectively. Only 1 developer each works in the areas of system software, documentation and "Other".

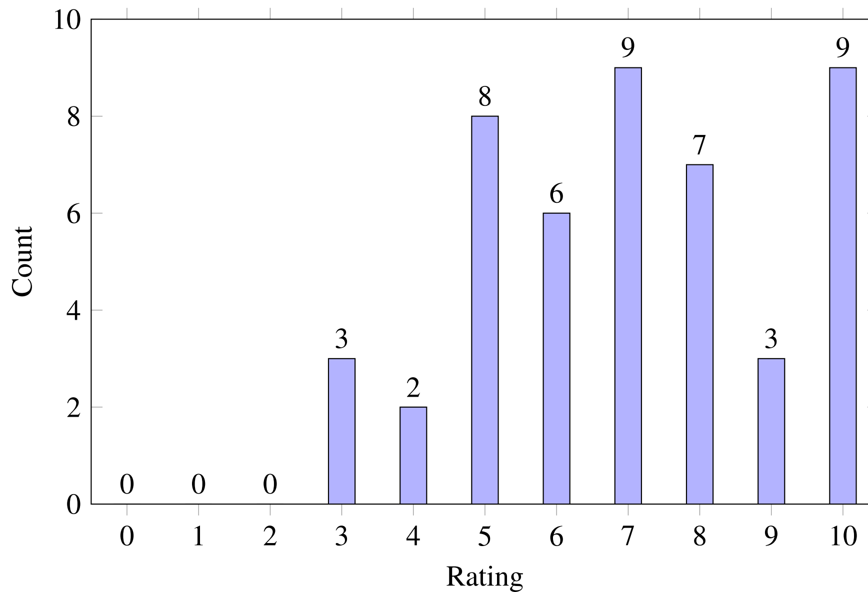

Figure 5.19: Reliance on Java built-in methods by surveyed Software Developers

We combined the responses from questions 5, 7, and 9, which aimed to determine the most frequently used methods for Java 8, 11, and 17, respectively. Table 5.5 showcases the most popular built-in methods from these Java versions.
The “stream()” method from Java 8 was chosen most often by respondents, by 98.0% of respondents. Meanwhile, 2.0% chose the “Other” category when none of the specified methods matched their experience. This 2.0% is a single developer who also noted that all of the methods listed in the question, with the exception of “stream()”, are very niche APIs.

Moving on to Java 11, the “HttpRequest.Builder header()” method stood out as the preferred choice for 39.0% of respondents. A close second, 35.0% of respondents selected the “Other” option, suggesting they favored methods not listed in the survey. The “HttpRequest.Builder setHeader()” method came in third, chosen by 22.0% of participants. Lastly, the “WebSocket.Builder header()” method garnered the attention of 4.0% of respondents. Notably, among those who chose the “Other” option, one developer specified their most-used method as “Files.writeString()”.

Lastly, for Java 17, the “Other” option was the most popular, chosen by 59.0% of respondents indicating they use methods not explicitly mentioned in the survey. This was followed by the “ByteBuffer put()” method, favored by 29.0% of respondents. The “mismatch()” method rounded out the top three, with a 12.0% usage rate. Interestingly, among the respondents who chose the “Other” category, two respondents mentioned their frequent use of features such as record and pattern matching.

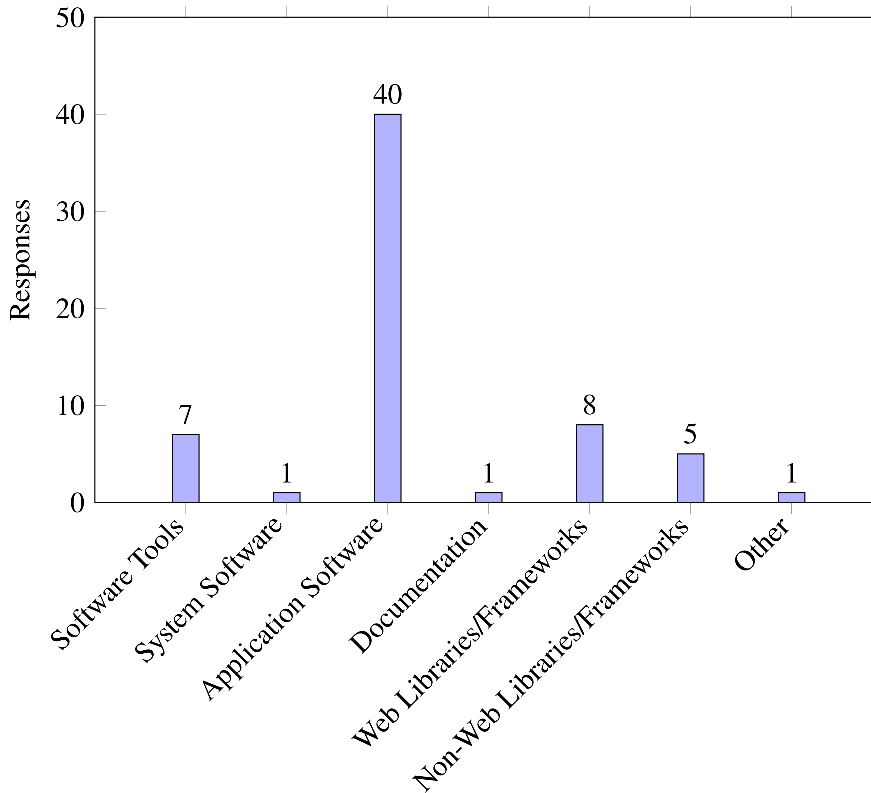

Figure 5.20: Primary Domain of Work of Software Developers

	1st Most Used	2nd Most Used	3rd Most Used	4 rd Most Used
Java 8	stream() 98.0%	Other 2.0%	–	–
Java 11	HttpRequest.Builder header() 39.0%	Other 35.0%	HttpRequest.Builder setHeader() 22.0%	WebSocket.Builder header() 4.0%
Java 17	Other 59.0%	ByteBuffer put() 29.0%	mismatch() 12.0%	–

Table 5.5: Most used Java built-in methods from Java 8, 11, and 17 by surveyed Software Developers

On the other hand, we have also the least used methods for Java 8, 11, and 17. As we did before we also combined the responses from questions 6, 8, and 10 in a unique table, where we can see the results. Table 5.6 showcases the least popular built-in methods from these Java versions.

For Java 8, the "Calendar.Builder setTimeOfDay()" method was identified as the least utilized. 44.0% of the respondents surveyed mentioned that they seldom or never use this method in their projects. "LongStream concat()" was the next on the list of infrequently used methods, as indicated by 27.0% of the respondents. Interestingly, the "stream()" method, despite being the most popular

method for Java 8 overall from the previous question, was reported as infrequently used by 7.0% of respondents.

In Java 11, the "WebSocket.Builder header()" method was identified as the least used by a majority of respondents, with 59.0% indicating they rarely resort to it. Notably, 30.0% selected the "Other" option, suggesting a variety of lesser-known or utilized methods that were not explicitly listed in the survey. The "HttpRequest.Builder setHeader()" and "HttpRequest.Builder header()" methods also emerged as infrequently used, with 7.0% and 4.0% of the respondents marking them as their third and fourth least-used methods, respectively.

For the most recent version, Java 17, the "mismatch()" method emerged as the least favored, with 50.0% of respondents indicating its infrequent use. Closely following, 28.0% of respondents selected the "Other" option, hinting at other methods in Java 17 they use even less. The "ByteBuffer put()" method was also highlighted as infrequently used, with 22.0% of respondents noting its lesser usage.

	1st Least Used	2nd Least Used	3rd Least Used	4 rd Least Used
Java 8	Calendar.Builder setTimeOfDay() 44.0%	LongStream concat() 27.0%	Calendar.Builder setDate() 16.0%	stream() 7.0%
Java 11	WebSocket.Builder header() 59.0%	Other 30.0%	HttpRequest.Builder setHeader() 7.0%	HttpRequest.Builder header() 4.0%
Java 17	mismatch() 50.0%	Other 28.0%	ByteBuffer put() 22.0%	–

Table 5.6: Least used Java built-in methods from Java 8, 11, and 17 by surveyed Software Developers

Questions 11 through 16 probed into the experiences of respondents regarding the identification of bugs and code smells by SonarQube post-Java version upgrades. Figure 5.21 shows a breakdown of Yes/No answers that provides information about the number of respondents who observed an increase in bugs or code smells detected by SonarQube when migrating to Java 8, 11, and 17 compared to previous Java versions. Meanwhile figure 5.22 provides similar insights but specifically for bugs.

In the first figure, highlighting the issue of code smell, we see that 5 developers noticed an increase in code smell when switching to Java 8, while another 5 did not notice any increase. When moving to Java 11, fewer respondents, 4, noticed an increase in code smell. However, 10 respondents reported no increase in such problems. For Java 17, the picture is mixed: 6 respondents noted an increase in code smells, but a slightly smaller group of 4 saw no significant change.

The second figure, which looks at bugs, shows that for Java 8, 3 respondents noticed an increase in bugs, while 7 noticed no change. For Java 11, the numbers were even more favorable: only 1 developer noticed an increase in bugs, while 7 noticed no difference. The trend changes slightly for Java 17, where 3 respondents noticed more bugs, but 6 noticed no significant increase.

Overall, the majority of respondents did not observe a significant increase in code smells or bugs detected by SonarQube after adopting any of the three Java versions.

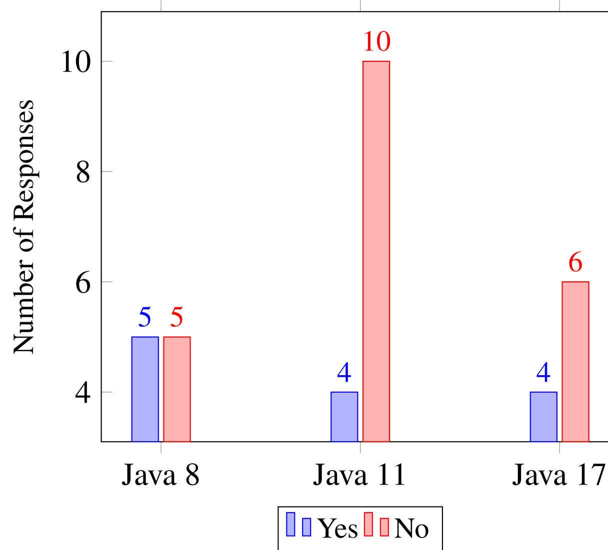

Figure 5.21: Increase in Code Smells Detected by SonarQube After Adopting Java 8, 11, and 17

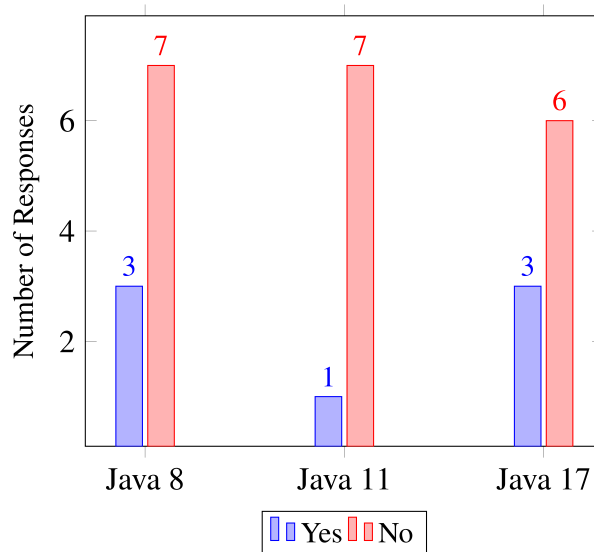

Figure 5.22: Increase in Bugs Detected by SonarQube After Adopting Java 8, 11, and 17

6 Discussion

In our analysis, based on data from the Table 5.1, we noticed an interesting trend: Java's built-in methods are underused. Despite Java's extensive list of built-in methods, they account for only a tiny fraction of the methods used by developers in open-source projects, just 0.18% of the total. Even if we remove the commonly used setters and getters, this number increases only slightly to 0.26%.
This result could be attributed to multiple reasons. With Java 17 boasting 37,584 built-in methods, some developers might struggle to stay updated on all the available options. For those new to Java, the vast array of native methods can be daunting. Some might be unaware of particular methods or deem them too intricate to implement.
In addition, the community and companies have developed numerous third-party libraries over the years. These libraries may be an attractive alternative for some developers because they are easier or better suited for their tasks. If developers have a strong familiarity with specific libraries, it is highly probable they will utilize the methods inherent to those libraries rather than trying to use a native Java method.
Another factor could be the influence of legacy code. Many projects on GitHub may have their roots in older versions of Java. As projects evolve and expand, developers often build on existing code rather than rewriting sections to incorporate newer built-in methods. The need for maintenance and the reluctance to upgrade could restrict the adoption of newer Java built-in methods, favoring stability over exploration.

In our dataset regarding the Java built-in methods, we can see the distinct dominance of methods from Java 6. Maybe this version brought in methods that many developers found useful. It is also possible that certain repositories, specialized in particular areas, heavily rely on Java 6 methods. Another reason could be that some third-party libraries were built using the methods from this version, making them popular.
However, it is curious to see a lack of adoption for newer Java versions, mainly Java 8 to 17. Perhaps these versions introduced features that did not find wide acceptance, or developers might be using third-party libraries that already implement these methods, reducing the need for them to use the methods directly. Alternatively, it could just be a matter of time before their acceptance increases.
The data from the projects shows that the "radiance" project provided 35.0% of the Java built-in methods we found. This single project might change the way we see the results. The fact that 33.0% of the projects (equivalent to 714) provided no data on the use of Java built-in methods suggests several possibilities: either they are not using Java built-in methods, our analysis missed these methods, or there are other underlying reasons we can only speculate about. The reason behind this lack could be influenced by the nature of the project, its age, or the personal preferences and experience of the developers.
When looking at RQ2, we need to think not only how many bug or code smell rules are associated with each Java version, but also how often each version is used in projects. If some versions do not

have many methods, are not popular with developers, or if projects are not updated to the latest version, then we will not see the use of new features. This means that some rules may not be triggered in tools like SonarQube.
Many bugs are linked to Java 1.1, 1.2, and 8. This could be because these versions have more methods or because more projects use them. On the other hand, the low bug counts in versions 1.4, 5, 6, 7, and 9 do not mean that they are better and that using methods from these versions could lead to fewer bugs. In the end, we had 0.0024 bugs per KLOC, which is a very low amount of bugs in the code. It might be that fewer projects use these versions or that they have fewer methods. If not many projects use a version, there is less chance to find this type of issues.
The high number of code smells compared to the bugs with Java 8 is also interesting. Maybe Java 8, with its new features, was used in many different types of projects. Projects with tricky code might have more code smells, especially if they are trying new ways of implementing functions. As for the bugs, we also have a minimal impact of code smells on the quality of the code, with 0.0324 code smells per KLOC.
Java 1.4 shows more "CRITICAL" problems than other versions, but fewer "MAJOR" ones. It could indicate that not many projects use methods or features from Java 1.4, or that it does not introduce many methods or features, but the ones it does introduce represent critical problems, if not fixed. Another aspect could be that SonarQube's rules are stricter for Java 1.4, resulting in more "CRITICAL" tags. In addition, factors other than methods, such as certain functions, can also influence the number of bugs or code smells.

In RQ3, we can see some clear trends in which Java built-in methods are preferred in different areas. The methods from Java 1.2 and 1.4 are chosen most often in areas such as "Application Software", "Documentation" and "System Software". This indicates that these versions introduced methods that are important or popular in these areas. Java 1.1 methods also seem to be quite popular.
On the other hand, methods from newer versions such as Java 8, 11, 12, and 16 are not chosen as frequently. There may be several reasons for this: Perhaps they are more specialized, or developers are still exploring these methods.
It is worth noting that methods from Java 6 appear most often in "Web Libraries and Frameworks". This is an indication that certain Java versions have introduced methods that are more suitable or beneficial for certain types of projects.
However, a consistent observation of the data is the minimal use of methods from versions such as 12 and 16, possibly because they are relatively new, unknown, or do not offer significant advantages over methods from earlier versions.
Observing the evolution of Java built-in method usage over the years, it is clear that developer behavior has changed. Periods of growth and decline in method usage could be related to broader trends in the technology industry, the introduction of competing languages, or changes in programming practices. The recent upward trend may indicate that developers are becoming more accustomed to, or even reliant on, the latest Java versions and their integrated offerings. Nevertheless, the number of Java built-in methods is relatively low compared to the total method usage. With 1.03 Java built-in methods per KLOC, we cannot say that the methods introduced in the current version of Java are popular among developers.
From our analysis, it becomes clear that while older Java versions like Java 1.0, 1.2, and 1.4 have consistent representation across varying project sizes and contributor counts, Java 6 stands out, particularly in the largest projects. Java 6 methods are notably prevalent, especially in the largest projects (75-100 percentile) and interestingly, also in projects falling within the 0-25 percentile

of contributors. The minimal representation of versions like Java 12 and 16 suggests that these versions might either be too new, lack methods that developers find valuable, or overlap with existing third-party libraries.

In addressing our final research question, we observed that certain responses were consistent with our findings regarding the adoption of Java built-in methods, particularly with respect to the most and least used methods. Interestingly, the survey revealed an average score of 7.6, indicating that participants in the survey relied in a good measure on Java built-in methods. This is somewhat unexpected and stands in stark contrast to our analysis of open-source projects, where Java built-in methods appear to have been underutilized. This difference could be due to the different backgrounds of survey participants compared to those who contribute to open-source projects on GitHub. Moreover, there are many complex projects on GitHub that may prefer custom solutions. In such projects, methods with extensive logic-and using Java built-in methods are used repeatedly, which explains the observed discrepancy. Similarly, smaller projects might rely heavily on predefined third-party libraries for implementation, bypassing the need for Java's built-in methods.
In terms of software quality, we found that code smells and bugs specific to a particular Java version did not have a significant impact on the overall quality of the software. These results are consistent with the survey responses, where the majority of participants indicated that they did not see a significant increase in code smells or bugs detected by SonarQube after adopting a specific Java version.

In summary, our analysis has shed light on the different patterns of adoption of Java built-in methods in different Java versions in open-source projects. Despite the richness of Java built-in methods, their adoption remains surprisingly low in the analyzed datasets, raising questions about the underlying factors that influence developers' choices. The discrepancy between survey responses and actual usage patterns in open-source projects also underscores the complex interaction of factors such as developer familiarity, project requirements, legacy considerations, and enterprise standards.
This research serves as a starting point for a deeper exploration of the dynamics of programming practices. The factors we have identified should be explored further. They may shed light on how language design, documentation, and developer training can be improved to promote optimal use of integrated methods.
Future studies could delve more deeply into the qualitative aspects of developer decision-making, using interviews or focus groups to provide nuanced insights. Similarly, tracking the adoption of integrated methods over time, especially with the introduction of newer Java versions, can provide a dynamic understanding of developer preferences and industry trends.

7 Threats to Validity

In any empirical study, we need to identify potential threats to the validity of the results and implement some steps to mitigate them. Given the large scale of our study and the complexity of implementation, several threats to validity may arise [Bal18]. In this section, we address these threats and the specific challenges we encountered. In addition, we explain the measures we took to mitigate or eliminate these threats wherever possible.

7.1 Internal Validity

The fourth research question, which delves into how the observed adoption of Java version features aligns with the personal experiences of Java developers, focuses on a survey involving 47 developers. The survey provided valuable perspectives, yet with a group of just 47, it is uncertain how closely it aligns with the broader Java developer population.
Individual experiences, backgrounds, and biases can significantly influence perceptions and opinions. For example, factors such as the specific projects an individual has worked on, the length of time they have been in the Java development world, or even regional development practices can influence their responses. This means that the feedback and insights from our survey are focused on the specifics of respondents' experiences rather than providing a universal perspective.
To mitigate some of these threats, we categorized the developers based on their experience with Java. We also chose close-ended questions to avoid ambiguity. Despite our efforts to find a representative number of developers, we encountered practical limitations. From a huge pool of potential participants, we received responses from only 47 individuals. Ideally, a more representative sample would consist of about 344 participants, given the size and diversity of the Java developer community. However, due to the exploratory nature of our study, these numbers should still be useful in gaining some insights, and the survey results can be used for validation in future studies.

Another threat to the internal validity comes from the analysis we performed on the temporal distribution of the different versions. The dynamic nature of software development, especially over an extended period of time such as 2011 to the present, presents inherent challenges to our research. Java, as a programming language, has undergone several updates, innovations, and shifts in coding standards. These external factors could affect the adoption of certain features of the Java version. One of the biggest dangers is that capturing these changes with an analysis may not be enough. Such a methodology might not fully account for variations in developer preferences, challenges, and inclinations over the years.
To understand the true nature of adoption trends of the various Java built-in methods, it would be ideal to consider the external events, such as the release of major software tools, programming paradigm shifts, like in Java 8, or even broader industry trends that might influence a developer's decision to adopt or ignore certain Java features.

For this reason, the choice of the time periods in which we analyzed the projects was not random, but programmed based on the release of the new Java versions. We tried to capture the status before and after the release to see if a significant difference could be detected and how often a particular method from a newer version would show up in our results.

Mapping methods to specific Java versions and designating them as Java built-in methods presented a particular challenge. If the implementation of the mapping process is not reliable enough, we could end up with a lot of inconsistent data. For example, we could miss many methods that are Java built-in methods but are not recognized by our implementation.
Our strategy for addressing these difficulties was based on a number of basic assumptions, each of which served as a level of review to ensure the accuracy of our classifications.
First, we considered the method names. If a method's name matched that of a known built-in Java method, it was kept for further validation. However, a name match alone was not sufficient to definitively classify a method.
The next level of validation dealt with parameter consistency. To ensure that a method was a true match, we verified that both the type and the number of parameters matched those of the potential Java built-in counterpart. For this check, we resorted to the "javaparser" library to help us identify the parameter type. While this library offers an extensive set of tools to parse and manipulate Java source code, its accuracy largely depends on the quality and consistency of the input. As with any automated tool, there is a potential for errors, especially when faced with edge cases or unconventional coding styles.
The last part of our methodology focused on the presence of libraries. We analyzed the list of libraries used by a class. The classification of a method as a Java built-in function depended on whether the associated library was present in this list. If the required library was missing or the parameters did not match, the method was not classified as built-in.
Due to the large amount of files and data we analyzed, there may have been some exceptions. Developers may use methods from libraries that they did not explicitly list, or the tool may occasionally misinterpret parameter types. Such anomalies, while rare, highlight the challenges of using a heuristic approach, even though we did our best to ensure a good level of accuracy.

To classify GitHub projects into specific domains, we utilized the GPT-3.5 model to process READMEs. Our choice of GPT-3.5 was motivated by its demonstrated prowess in handling varied and domain-specific content, an inherent ability to discern context without task-specific training, and its superior performance in categorization as seen in numerous studies [AAA+23] [KWNA23] [Lam23]. However, as powerful as GPT-3.5 may be, it is not infallible. Possible biases due to training data or misclassification could affect the conclusions of our research. To address this, we manually validated the model's classifications for 10.0% of our sample, which yielded a commendable 96.0% accuracy rate.

7.2 External Validity

We used GitHub, a platform that hosts projects from developers worldwide, as the main source for our research. This means that not every developer uses the same coding standards. Educational background can also influence coding practices, preferences for certain Java features, and even perceptions of software quality. Also, since we do not know which country the projects come from, we cannot generalize the results of our study. Our sample might be geographically diverse, representing multiple countries and regions, or it might be biased towards certain locations. We have not included this validation step in our implementation due to the challenges that such validation presents.
While we made a conscious effort to select projects based on objective criteria that have been analyzed and suggested in previous mining studies, such as star ratings, activity levels, and a predominance of Java code (greater than 75.0%), these criteria might unintentionally correlate with repositories from certain geographic areas or cultural groups. For example, very active or high scoring repositories might be more prevalent in regions with a stronger open-source community or in regions where Java is a predominant programming language in academia and industry. Such unintended biases could limit the generalizability of our results.

Another aspect to consider is that we relied on web scraping techniques to obtain information on Java built-in methods from the official Oracle documentation. Although web scraping inherently raises concerns about data accuracy, we took extra measures to ensure the reliability of our dataset. Our scraping tool was designed to provide errors if it detected any inconsistencies with the scraping process and also to stop and write the error into a file if the site structure deviated from our expectations. To increase accuracy in categorizing Java built-in methods by their respective versions, we analyzed the method descriptions.
If a particular version was not apparent at the method level, we consulted the class description. In cases where the class itself did not contain version information, we assigned the method to the base version of Java, 1.0. This thorough methodology greatly reduces the potential for inaccuracies, although there may still be minor discrepancies. Still, potential discrepancies between our scraped dataset and the actual Oracle documentation could arise, whether due to changes in the site structure, limitations of our scraping tool, or other factors. Consequently, if such discrepancies exist, it may limit the generalizability of our findings. Our results, in that scenario, might not comprehensively or accurately represent the full scope of Java built-in methods as documented by Oracle.
However, we believe that our results, also based on the data obtained in previous studies about the number of methods built into Java [MAT06][ANMT11], largely reflect the information contained in Oracle's documentation, which supports the generalizability of our results.

7.3 Construction Validity

In our examination of the second research question concerning software quality, we centered our analysis on two significant indicators: code smells and bugs. Although these indicators provide valuable insights into certain aspects of software quality, it is important to specify that they do not capture the totality of software quality.

Software quality is inherently multi-faceted, encompassing dimensions such as functionality, maintainability, usability, reliability, performance, and security, to name a few [SWC10]. We also have various models and standards that have been proposed over the years to capture and define software quality in a structured manner [1] [ORe11]. The approach we have taken primarily emphasizes the maintainability aspect, which is reflected in the analysis of code smells [2] [YM12b]. Bugs provide insight into software reliability [3]. The selected set of 72 rules from SonarQube (24 regarding bugs and 46 for code smells) provides a specific perspective through which software quality is viewed. While this provides detailed insight into specific aspects of quality, some quality attributes may be underrepresented or overlooked.
Considering these possible limits, we have clearly set out what we wanted to research and explore "Software Quality" in our research. We emphasize that our findings primarily capture maintainability, touch on certain facets of reliability, and provide a limited perspective on security. For a more comprehensive view of software quality in future studies, it would be beneficial to integrate different data sources such as performance metrics, user feedback, or crash reports specific to a java version feature. This could provide a more comprehensive understanding and a richer contextual background.

In our research, SonarQube rules were manually mapped to specific Java versions. This manual classification inherently poses a threat to conclusion validity due to the potential for human errors. While every effort was made to ensure accuracy, subjective interpretations could influence the categorization. To mitigate this risk and enhance the reliability of our findings, we cross-referenced the rule descriptions with multiple authoritative sources, including books [Sha18] and Oracle documentation. This ensured that any association of a rule to a particular Java version was grounded in the presence of a specific method or feature unique to that version. Despite these measures, it is important to approach our conclusions with awareness of the manual categorization process and its potential limitations.

7.4 Conclusion Validity

To map methods to specific Java versions, we extracted data from Oracle documentation using web scraping techniques. While the "BeautifulSoup" library facilitated access to the HTML content of the website, one should note its potential limitations. Although this library excels at parsing standard HTML, non-standard or irregular HTML structures can be challenging. Furthermore, the library relies on the consistent structure of the webpage; changes or updates in the layout can interfere with data extraction.
Since we relied heavily on the scraped data, the reliability and consistency of the scraping process were of major importance. Any inconsistency in this process could compromise the credibility of our conclusions. Such inconsistencies could introduce uncertainty and potentially bias our results. To address this, we conducted rigorous testing of our implementation. Since our data is HTML content, careful manual testing was required to understand the format and to implement the logic. The script needed to reliably determine the version of a given Java built-in method. In manually

[1] ISO/IEC 25010 https://iso25000.com/index.php/en/iso-25000-standards/iso-25010
[2] Maintainability https://iso25000.com/index.php/en/iso-25000-standards/iso-25010/57-maintainability
[3] Reliability https://iso25000.com/index.php/en/iso-25000-standards/iso-25010/62-reliability

testing numerous methods, we noticed a consistent pattern: in the description of each method was indicated the version in which it was introduced. If this version information was missing, the method inherited its version from its parent class. However, if this version information was missing for a class, it was labeled “Default” by us, corresponding to version 1.0. This decision was based on our observations that classes without explicit version information usually matched the original 1.0 version of Java.
We may have some false positives, i.e. methods labeled with version 1.0 are from a different version, but this case should be rare and should not pose a risk to our analysis. For this reason, we believe that this risk should not significantly affect the results of our analysis.

For our analysis and parsing of the Java code in order to extract the needed data, we used an external tool, namely javaparser. Using javaparser as the main tool for parsing Java code has potential challenges. Every software library, including javaparser, has its inherent limitations. It is possible that there are certain Java constructs or methods that the library does not parse with extreme precision or even misses altogether. For this reason, we performed a manual review of a selected subset of Java files during our testing phase to verify the accuracy of method extraction. However, given the large number of files we processed, the preliminary tests may not be sufficiently extensive to ensure proper parsing of the library in all scenarios. We see this as a starting point for future research, where improved validation procedures could further mitigate this risk.

8 Conclusion

As part of our research, we conducted an extensive data mining study supplemented by quantitative survey findings. We obtained our data from a combination of sources. Firstly, we sourced from open-source Java projects on GitHub, and secondly, we utilized the documentation of Java built-in methods from Oracle's official documentation. As part of our analysis, we examined 2,167 repositories, focusing on three known LTS Java versions, namely 8, 11, and 17, which have 34,676, 34,916, and 37,584 Java built-in methods, respectively.
This research highlighted the current adoption of Java built-in methods in open source projects. In addition, we used a survey to capture developers' views on Java built-in Methods and the different Java versions, where a total of 47 developers took part.

From our results, we observed a minimal overall adoption of Java built-in methods, averaging 1.03 methods per KLOC. Notably, 750 projects did not utilize a single Java built-in method. However, there was a pronounced usage of methods from Java 6, with a contrasting decline in their adoption from Java 8 through 17. This is a discrepancy when we consider the results of the survey, in which developers indicated that the reliance on these methods was pretty high, with an average of 7.6 on a scale of 1 to 10, where 1 indicates "Very rarely" and 10 indicates "Very often". We also saw how the popularity of some methods like "stream()" were validated from both results, from the data we collected and from the developers' responses.
In terms of software quality, we also saw that bugs and code smells related to a particular version have a minimal impact on the overall quality of a software, with an average of 0.0025 bugs per KLOC and 0.0324 code smells per KLOC. The survey also indicates that the majority of the participants did not see a significant increase in code smells and bugs when moving from an older Java version to a newer one.
Our analysis found that domain selection generally did not significantly influence the adoption of Java built-in methods. However, a notable exception was observed in the "Web Libraries and Frameworks" domain. Within this domain, there was a 10.0% lower usage rate of methods from version 1.2 compared to their rate of use in other domains. Moreover, there was a substantial increase in the usage of methods introduced in Java 6, rising from a range of 3.0-6.0% in other domains to 20.0% for this specific domain.

It is worth noting that our study has some limitations: our survey relied primarily on open-source projects, which may not be fully representative of proprietary enterprise projects, and survey respondents may not represent the larger Java developer community. Based on the results of our study, we see potential for future research to further validate and investigate the adoption of Java features.
While our study focused primarily on Java built-in methods, third-party libraries also play a significant role in software projects. One interesting future work would be to analyze software projects from different companies in the software development field. Some companies may maintain their own coding standards and show varying degrees of reluctance or preference in integrating third-party libraries. By examining these projects, one could gain a more detailed understanding of

the patterns of Java feature adoption across the industry. This in-depth investigation could reveal whether enterprise coding practices and policies affect the nature of feature adoption in a significant way.
Another future research could be the use of SonarQube to evaluate the quality of projects that have a long history and have been upgraded to a newer version of Java during their lifecycle. SonarQube is able to highlight potential coding issues related to a certain Java version. By focusing on projects that have been upgraded to a newer Java version, one can examine the impact of such upgrades on the overall quality of the code.
As Java continues to evolve and introduce a number of features and best practices, it would be beneficial to examine how these innovations play out in practice and what impact they have on code quality.
Lastly, future research could benefit from in-depth interviews with open questions. The incorporation of open-ended question interviews could provide deeper insight into the adoption trends of Java built-in methods and compare them with the results we obtained in this research. By engaging directly with developers, it is possible to explore the motivations, challenges, and influences that affect a specific Java feature decision.
In addition, these interviews can serve as a validation mechanism. While the data obtained from this research may point to certain trends, talking directly with practitioners can confirm or refute these findings. For example, if the data from our research indicates a low adoption rate for a particular Java feature, an interview might reveal that this is due to specific complexity, a lack of adequate documentation, or even external factors such as team or company policies.

In summary, our research provides important insight into the evolving dynamics of Java built-in method adoption in open-source projects, revealing both expected and unexpected patterns. The discrepancies between actual usage, developer perceptions, and domain-specific variations underscore the complicated landscape of software development. As Java remains central to the industry, understanding these nuances not only provides insight into current practices, but also helps shape strategies and tools for future Java iterations and the broader software community.

Bibliography

[AAA+23] Z. Alyafeai, M. S. Alshaibani, B. AlKhamissi, H. Luqman, E. Alareqi, A. Fadel. "Taqyim: Evaluating Arabic NLP Tasks Using ChatGPT Models". In: *arXiv (Cornell University)* (2023). DOI: 10.48550/arxiv.2306.16322. URL: http://arxiv.org/abs/2306.16322 (cit. on pp. 34, 68).

[ANMT11] C. Anslow, J. Noble, S. Marshall, E. Tempero. "Visualizing the Size of the Java Standard API". In: (May 2011). URL: https://www.researchgate.net/publication/229002555_Visualizing_the_Size_of_the_Java_Standard_API (cit. on pp. 22, 23, 69).

[Bal18] L. Baldwin. *Research concepts for the practitioner of educational leadership*. June 2018. DOI: 10.1163/9789004365155. URL: https://doi.org/10.1163/9789004365155 (cit. on p. 67).

[BHV16] H. Borges, A. Hora, M. T. Valente. "Understanding the Factors That Impact the Popularity of GitHub Repositories". In: (2016), pp. 334–344. DOI: 10.1109/ICSME.2016.31 (cit. on p. 33).

[Cln13] L. I. S. Cln. "Data collection techniques a guide for researchers inhumanities and education". In: *International Research Journal of Computer Science and Information Systems* 2 (Apr. 2013), pp. 40–44. URL: https://www.academia.edu/490747/Data_collection_techniques_a_guide_for_researchers_in_humanities_and_education (cit. on p. 37).

[CS15] F. Chatziasimidis, I. Stamelos. "Data collection and analysis of GitHub repositories and users". In: (2015), pp. 1–6. DOI: 10.1109/IISA.2015.7388026 (cit. on p. 26).

[DLT19] T. T. Dien, B. H. Loc, N. Thai-Nghe. "Article Classification using Natural Language Processing and Machine Learning". In: (2019), pp. 78–84. DOI: 10.1109/ACOMP.2019.00019 (cit. on p. 33).

[DSG+22] J. Dykema, N. C. Schaeffer, D. Garbarski, N. Assad, S. Blixt. "Towards a reconsideration of the use of agree-disagree questions in measuring subjective evaluations". In: *Research in Social and Administrative Pharmacy* 18.2 (Feb. 2022), pp. 2335–2344. DOI: 10.1016/j.sapharm.2021.06.014. URL: https://doi.org/10.1016/j.sapharm.2021.06.014 (cit. on p. 38).

[GGTF15] P. Gyimesi, G. Gyimesi, Z. Tóth, R. Ferenc. "Characterization of Source Code Defects by Data Mining Conducted on GitHub". In: (2015). Ed. by O. Gervasi, B. Murgante, S. Misra, M. L. Gavrilova, A. M. A. C. Rocha, C. Torre, D. Taniar, B. O. Apduhan, pp. 47–62 (cit. on p. 26).

[Kan14] S. H. Kan. *Metrics and Models in Software Quality Engineering - Paperback*. 2nd. Addison-Wesley Professional, 2014. ISBN: 0133988082 (cit. on p. 40).

[KGB+15] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. German, D. Damian. "An in-depth study of the promises and perils of mining GitHub". In: *Empirical Software Engineering* 21 (Sept. 2015). DOI: 10.1007/s10664-015-9393-5 (cit. on p. 26).

[KWNA23] M. T. I. Khondaker, A. Waheed, E. M. B. Nagoudi, M. Abdul-Mageed. "GPTARAEVaL: A Comprehensive Evaluation of ChatGPT on Arabic NLP". In: *arXiv (Cornell University)* (May 24, 2023). DOI: 10.48550/arxiv.2305.14976. URL: http://arxiv.org/abs/2305.14976 (cit. on pp. 34, 68).

[Lam23] B. Lamichhane. "Evaluation of CHATGPT for NLP-based Mental Health Applications". In: *arXiv (Cornell University)* (Mar. 28, 2023). DOI: 10.48550/arxiv.2303.15727. URL: http://arxiv.org/abs/2303.15727 (cit. on pp. 34, 68).

[MAT06] H. Ma, R. Amor, E. Tempero. "Usage Patterns of the Java Standard API". In: (2006), pp. 342–352. DOI: 10.1109/APSEC.2006.60 (cit. on pp. 13, 21, 23, 69).

[Mil96] J. Miller. In: *Politics and the Life Sciences* 15.1 (1996), pp. 139–141. ISSN: 07309384, 14715457. URL: http://www.jstor.org/stable/4236218 (visited on 10/26/2023) (cit. on p. 37).

[ORe11] G. O'Regan. *Capability Maturity Model IntegrationCapability Maturity Model Integration*. London: Springer London, 2011, pp. 43–65. ISBN: 978-0-85729-172-1. DOI: 10.1007/978-0-85729-172-1_3. URL: https://doi.org/10.1007/978-0-85729-172-1_3 (cit. on p. 70).

[PBM12] C. Parnin, C. Bird, E. Murphy-Hill. "Adoption and use of Java generics". In: *Empirical Software Engineering* 18.6 (Dec. 2012), pp. 1047–1089. DOI: 10.1007/s10664-012-9236-6. URL: https://doi.org/10.1007/s10664-012-9236-6 (cit. on pp. 13, 21).

[QGB+17] P. Quezada Sarmiento, D. Guaman, L. R. Barba Guamán, L. Enciso, P. Cabrera. "SonarQube as a tool to identify software metrics and technical debt in the source code through static analysis". In: (July 2017) (cit. on p. 32).

[QLL16] D. Qiu, B. Li, H. Leung. "Understanding the API usage in Java". In: *Information Software Technology* 73 (May 2016), pp. 81–100. DOI: 10.1016/j.infsof.2016.01.011. URL: https://doi.org/10.1016/j.infsof.2016.01.011 (cit. on pp. 22, 29).

[RM16] C. Robson, K. McCartan. *Real world research*. Jan. 2016. URL: http://eprints.uwe.ac.uk/27650/ (cit. on p. 14).

[RMN19] J. Rashid, T. Mahmood, M. Nisar. "A Study on Software Metrics and its Impact on Software Quality". In: (May 2019). URL: https://www.researchgate.net/publication/333505554_A_Study_on_Software_Metrics_and_its_Impact_on_Software_Quality (cit. on p. 40).

[RPDF17] B. Ray, D. Posnett, P. Devanbu, V. Filkov. "A Large-Scale Study of Programming Languages and Code Quality in GitHub". In: *Commun. ACM* 60.10 (Sept. 2017), pp. 91–100. ISSN: 0001-0782. DOI: 10.1145/3126905. URL: https://doi.org/10.1145/3126905 (cit. on p. 23).

[SBLR19] N. Saarimaki, M. T. Baldassarre, V. Lenarduzzi, S. Romano. "On the Accuracy of SonarQube Technical Debt Remediation Time". In: (2019), pp. 317–324. DOI: 10.1109/SEAA.2019.00055 (cit. on p. 32).

[Sch11] D. J. Schaeffer NC. "Questions for Surveys: Current Trends and Future Directions." In: *Public Opin Q. 2011 Dec;75(5):909-961.* (2011). DOI: 10.1093/poq/nfr048.. URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4071995/ (cit. on p. 38).

[Sha18] K. Sharan. *Java Language Features*. 2018. DOI: https://doi.org/10.1007/978-1-4842-3348-1. URL: https://link.springer.com/book/10.1007/978-1-4842-3348-1#bibliographic-information (cit. on p. 70).

[SMLG17] V. Saradhi, M. Manisha, L. Lakshmi, K. Gowtham. "Bug reduction with reliability factors using hybrid reliable model". In: *International Journal of Engineering Technology* 7 (Dec. 2017), p. 397. DOI: 10.14419/ijet.v7i1.1.9860 (cit. on p. 19).

[Sul20] M. Sulír. "String Representations of Java Objects: An Empirical Study". In: *SOFSEM 2020: Theory and Practice of Computer Science*. Ed. by A. Chatzigeorgiou, R. Dondi, H. Herodotou, C. Kapoutsis, Y. Manolopoulos, G. A. Papadopoulos, F. Sikora. Cham: Springer International Publishing, 2020, pp. 479–490. URL: https://link.springer.com/chapter/10.1007/978-3-030-38919-2_39 (cit. on p. 23).

[SV02] J. Singer, N. Vinson. "Ethical issues in empirical studies of software engineering". In: *IEEE Transactions on Software Engineering* 28.12 (2002), pp. 1171–1180. DOI: 10.1109/TSE.2002.1158289 (cit. on p. 37).

[SWC10] D. Samadhiya, S.-H. Wang, D. Chen. "Quality models: Role and value in software engineering". In: 1 (2010), pp. V1-320-V1–324. DOI: 10.1109/ICSTE.2010.5608852 (cit. on p. 70).

[SYA+13] D. I. Sjøberg, A. Yamashita, B. C. Anda, A. Mockus, T. Dybå. "Quantifying the Effect of Code Smells on Maintenance Effort". In: *IEEE Transactions on Software Engineering* 39.8 (2013), pp. 1144–1156. DOI: 10.1109/TSE.2012.89 (cit. on p. 19).

[Wag13] S. Wagner. *Software Product Quality Control*. Jan. 2013. DOI: 10.1007/978-3-642-38571-1. URL: https://doi.org/10.1007/978-3-642-38571-1 (cit. on p. 18).

[WCH+20] Y. Wang, B. Chen, K. Huang, B. Shi, C. Xu, X. Peng, Y. Wu, Y. Liu. "An Empirical Study of Usages, Updates and Risks of Third-Party Libraries in Java Projects". In: (2020), pp. 35–45. DOI: 10.1109/ICSME46990.2020.00014 (cit. on p. 22).

[YM12a] A. Yamashita, L. Moonen. "Do code smells reflect important maintainability aspects?" In: (2012), pp. 306–315. DOI: 10.1109/ICSM.2012.6405287 (cit. on p. 19).

[YM12b] A. Yamashita, L. Moonen. "Do code smells reflect important maintainability aspects?" In: (2012), pp. 306–315. DOI: 10.1109/ICSM.2012.6405287 (cit. on p. 70).

[ZWZG17] R. Zhou, X. Wang, S. Zhang, H. Guo. "Who tends to answer open-ended questions in an e-service survey? The contribution of closed-ended answers". In: *Behaviour Information Technology* 36.12 (Sept. 2017), pp. 1274–1284. DOI: 10.1080/0144929x.2017.1381165. URL: https://doi.org/10.1080/0144929x.2017.1381165 (cit. on p. 37).

All links were last followed on 2. November, 2023.