Thesis for the degree of Master of Science

Comparative Analysis of Cross-Platform and Native Mobile App Development Approaches

Ibrokhimov Sardorbek Rustam Ugli

Department of Information Convergence Engineering
The Graduate School
Pusan National University

August 2024

Comparative Analysis of Cross-Platform and Native Mobile App Development Approaches

A Thesis submitted to the graduate school of Pusan National University in partial fulfillment of the requirements for the degree of Master of Science under the direction of Gyun Woo

The thesis for the degree of Master of Science

by Ibrokhimov Sardorbek Rustam Ugli

has been approved by the committee members.

May 31, 2024

Chair	Hwan-Gue Cho	__________
Member	Heung-Seok Chae	__________
Member	Gyun Woo	__________

Contents

I Introduction 1

1.1 The Rise of Mobile Applications 1

1.2 Evolution of Mobile Development Approaches 3

1.3 Importance of Choosing the Right Development Approach 3

1.4 Comparative Analysis: Java, Kotlin, and Flutter 4

1.5 Critical Role of Code Quality 5

1.6 Research Gap 6

1.7 Study Objectives and Research Questions 7

1.8 Methodology, Significance, and Structure of Thesis 10

II Literature Review 12

2.1 Java 12

2.2 Kotlin 13

2.3 Dart(Flutter) 14

2.4 Previous Research 14

III Methodology 24

3.1 Overview 24

3.2 Identical Mobile Application Development 25

3.2.1 Project Planning. 25

3.2.2 Flutter Application Development 27

3.2.3 Kotlin Application Development 29

3.2.4 Java Application Development 31

3.3 SonarQube Code Inspection and Open Source Projects 33

3.3.1 Issue Severity in SonarQube 34

3.3.2 Setting Up SonarQube Environment 35
3.3.3 Applying SonarQube Analysis to the Kanban Board Project 36
3.3.4 Setting Up an Automated Pipeline for SonarQube Analysis of Open Source Projects 38
IV Results of Comparative Analysis 40
4.1 Overview 40
4.2 Results from Identical Application Development 40
4.2.1 Code Quality Analysis 41
4.2.2 Task Model Implementation 41
4.2.3 Application Size and Performance 42
4.3 SonarQube Implementation in Open Source Projects 42
4.3.1 Java Projects Analysis 42
4.3.2 Kotlin Project Analysis 43
4.3.3 Flutter Projects Analysis 43
4.3.4 Comparative Graph and Discussion 45
V Discussion 46
5.1 Interpretation of Results 46
5.2 Recommendations for Future Development Practices 47
5.3 Proposed Enhancements for Future Research 47
5.4 Reflection on Methodological Approaches 48
VI Conclusion 49
I Project Listings for Automated SonarQube Analysis 59
II Python Script for Project Automation and Configuration 62
III Python Script for Data Retrieval and Report Generation 67
IV Implementation Details of Task Models 71

List of Figures

1.1 Number of mobile app downloads worldwide from 2019 to 2027, by segment(in million downloads) [1] 2
1.2 Cross-platform mobile frameworks used by software developers worldwide from 2019 to 2022 [2] 8
2.1 Corral and Fronza's research resulted in a Market Success Model Based on App Store Metrics. 20
3.1 Planned view of Kanban Board Application. Designed Using Figma 27
3.2 Flutter version of the application's project structuring 28
3.3 Kotlin version of the application's project structuring 30
3.4 Java version of the application's project structuring 32
3.5 SonarQube Dashboard Overview: Code Quality Metrics for Kanban Board Projects in Flutter, Kotlin, and Java 37
4.1 Severity Distribution in Java Projects 43
4.2 Severity Distribution in Kotlin Projects 44
4.3 Severity Distribution in Dart Projects 44
4.4 Severity Distribution of Code Smells Across Languages 45

List of Tables

4.1 Severity levels of code smells detected in each language. 41
1.1 Java Open Source Projects Selected for Analysis 59
1.2 Kotlin Open Source Projects Selected for Analysis 60
1.3 Dart Open Source Projects Selected for Analysis 61

List of Abbreviations

SDK	Software Development Kit
API	Application Programming Interface
LOC	Lines of Code
UI	User Interface
HTTP	Hypertext Transfer Protocol
MVVM	Model-View-ViewModel
XML	eXtensible Markup Language
NOD	Number of Downloads
NOR	Number of Reviewers
AR	Application Ranking
CSV	Coma Separated Values
BLOC	Business Logic Component

Comparative Analysis of Cross-Platform and Native Mobile App Development Approaches

Ibrokhimov Sardorbek Rustam Ugli

Department of Information Convergence Engineering

The Graduate School

Pusan National University

Abstract

Though many approaches to developing mobile applications have been suggested up to now, developers have difficulties selecting the right one. This study compares native and cross-platform application development approaches, particularly focusing on the shift in preference from Java to Kotlin and the increasing use of Flutter. This research offers practical insights into factors influencing developers' choice of programming languages and frameworks in mobile application development by creating identical applications using Java, Kotlin, and Dart (Flutter). Furthermore, this study explores the best practices for development by examining the quality of code in 45 open-source GitHub repositories. The study evaluates LOC and code smells using semi-automated SonarQube assessments, including the measurement of severity levels, to determine the effects of selecting a specific language or framework on code maintainability and development efficiency. Preliminary findings show differences in the quality of the code produced by the two approaches, offering developers useful information on reducing code smells and improving project maintainability.

I Introduction

1.1 The Rise of Mobile Applications

Mobile applications have become integral to daily life globally, revolutionizing various sectors such as healthcare, education, finance, and entertainment. The rise of mobile applications has significantly impacted society, leading to increased dependence on mobile technologies. Integrating information and communication technologies (ICTs) like mobile phones and the Internet with financial services has given rise to new forms of digital finance, including mobile payments, online credit, and intelligent investment advice [3]. This integration has transformed financial services and facilitated access to formal financial services for underserved populations, such as smallholder farmers [4].

The rapid growth of the mobile app market is evident in the increasing adoption of mobile payment systems. Mobile payment systems have increased employment and income for family members, particularly benefiting low-income and rural households [5]. Moreover, the development of financial innovations like mobile payments and AI-based credit scoring systems has immense potential to increase financial inclusion, enabling the unbanked population to participate actively in financial markets [6].

In addition to finance, mobile applications have also played a crucial role in healthcare. Mobile smartphone applications provide a unique platform to promote the utilization of evidence-based skills, especially in areas like substance use treatment [7]. Furthermore, the education sector has witnessed a transformation with the increasing use of mobile applications. The study indicates that mobile banking, a mobile application, has provided smallholder farmers access to formal financial services like deposits and loans, which previously needed to be improved [4].

Entertainment is another sector significantly impacted by mobile applications. Note that mobile phones, through various entertainment apps, compete for limited attention with social networks, videos, games, and other types of entertainment [8]. This competition for attention underscores the widespread use and popularity of entertainment apps on mobile devices.

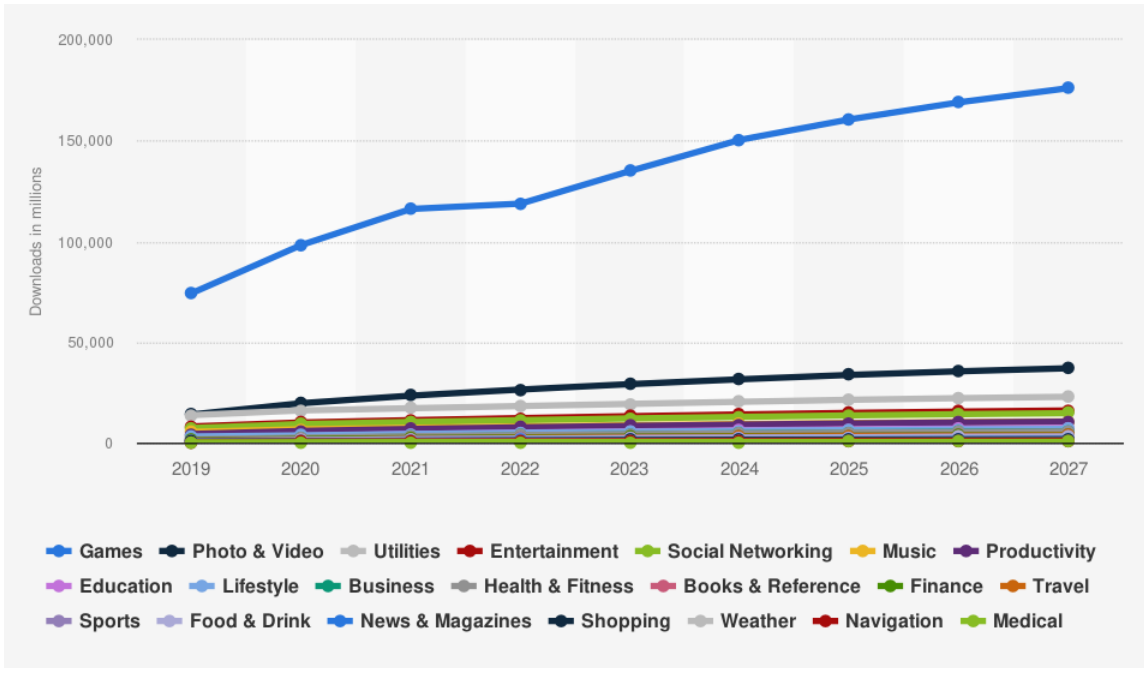


Figure 1.1: Number of mobile app downloads worldwide from 2019 to 2027, by segment(in million downloads) [1]

The Figure 1.1 clearly illustrates a continuous and significant growth in mobile application downloads across all sectors, with projections reaching up to 176.1 billion downloads in the Games segment by 2027. This trend, consistent throughout the forecast period from 2019 to 2027, emphasizes the expanding role and importance of mobile applications in daily life and various industries.

The statistics and trends in mobile application usage globally reflect a trajectory of growth and innovation. The increasing reliance on mobile technologies across sectors underscores the transformative impact of mobile applications on society, paving the way for enhanced accessibility, efficiency, and convenience in various aspects of daily life.

1.2 Evolution of Mobile Development Approaches

The evolution of mobile app development has seen significant advancements from early platforms to the modern ecosystems of Android and iOS. Initially, Java dominated Android development, offering a robust and versatile language for creating mobile applications. However, the introduction of Kotlin by JetBrains marked a pivotal moment in Android development. Kotlin, with its concise syntax, enhanced safety features, and seamless interoperability with Java, quickly gained popularity among developers. Google recognized the potential of Kotlin and endorsed it as a preferred language for Android app development, leading to widespread adoption within the Android community [8].

Moreover, the emergence of cross-platform frameworks has revolutionized mobile app development by enabling developers to write code once and deploy it across multiple platforms. Flutter, developed by Google, has gained prominence among these frameworks for its efficiency and flexibility in building high-quality native interfaces. Flutter allows for fast development, expressive UIs, and native performance, making it popular for developers aiming to create visually appealing and responsive applications.

Adopting Kotlin and Flutter signifies a shift towards more efficient and streamlined mobile app development practices. Kotlin's modern features and seamless integration with existing Java codebases have simplified the development process. At the same time, Flutter's cross-platform capabilities have empowered developers to create visually rich and performant applications across different operating systems. These advancements highlight the continuous evolution of mobile development approaches, emphasizing the importance of innovation and adaptability in the ever-changing landscape of mobile technology.

1.3 Importance of Choosing the Right Development Approach

Choosing between native development and cross-platform frameworks is a strategic decision that goes beyond technical considerations and impacts the success of a mobile application. Native development offers optimized performance and better access to device-specific features, ensuring high user satisfaction. On the other hand, cross-platform frameworks provide advantages such as faster rollout and cost-effectiveness. The decision between these approaches has implications for app performance, maintainability, and overall user experience, making selecting the appropriate development approach crucial.

Native development, often associated with languages like Java for Android and Swift for iOS, allows developers to leverage platform-specific features and functionalities, resulting in high-performance applications tailored to each operating system. However, the development time and cost for native apps can be higher due to the need to write separate codebases for different platforms.

In contrast, cross-platform frameworks like Flutter, React Native, and Xamarin enable developers to write code once and deploy it across multiple platforms, reducing development time and costs. While cross-platform development offers advantages in terms of efficiency and cost-effectiveness, there may be trade-offs in terms of performance optimization and access to certain platform-specific features.

The study emphasizes the importance of well-defined technical requirements and specifications in selecting a cross-platform framework or development approach. Specific cross-platform frameworks can perform equally or even better than native in particular metrics, highlighting the need for a deliberate and informed decision-making process [9].

Ultimately, the choice between native development and cross-platform frameworks should align with the mobile application's project requirements, budget constraints, and long-term goals. Understanding the implications of each approach on app performance, maintainability, and user satisfaction is essential for making an informed decision that maximizes the mobile application's success.

1.4 Comparative Analysis: Java, Kotlin, and Flutter

Java and Kotlin have been long-standing choices for native Android app development, with Java being the traditional standard and Kotlin emerging as a modern alternative.

Java, known for its robustness and extensive ecosystem, has been widely used for Android development [10]. On the other hand, Kotlin, being interoperable with Java, offers a more concise syntax and additional safety features, enhancing code readability and reducing bugs [11]. The transition from Java to Kotlin has been notable, with Google endorsing Kotlin as the preferred language for Android app development [12].

In contrast, Flutter, utilizing Dart, has gained attention for its cross-platform capabilities. It offers a single codebase for both Android and iOS platforms. This approach dramatically simplifies development and reduces maintenance costs [13]. Flutter has been recognized as a framework that streamlines the development of cross-platform applications, providing efficiency and cost-effectiveness [14].

The comparative analysis of Java, Kotlin, and Flutter reveals distinct advantages and challenges associated with each technology. Java and Kotlin excel in native Android development, with Kotlin offering modern features and improved safety. On the other hand, Flutter stands out for its cross-platform capabilities, enabling developers to create applications for multiple platforms with a single codebase. Understanding the strengths and limitations of each technology is essential for making informed decisions in mobile app development.

1.5 Critical Role of Code Quality

The quality of code is a vital component of software development, as it has a significant impact on the performance, scalability, and maintainability of applications. Key indicators of code quality include identifying code smells, which are patterns in the code that may indicate potential issues, and assessing the overall complexity of the codebase. Although code smells may not immediately affect the program's functionality, they can increase the risk of bugs or failures down the line and make maintenance more complex. Thus, it is critical to evaluate and address these smells to ensure the long-term quality and sustainability of software applications, as noted by Banker et al. [15].

In software development, code quality is closely tied to developers' skills and capabilities. Research has shown that content software developers are more productive and effective in problem-solving. Psychological measurements in empirical software engineering have emphasized the significance of factors such as high analytical problem-solving skills and creativity in the software construction process, highlighting the role of developer well-being in software quality and productivity [16].

Efforts to enhance software quality often involve methodologies like the Capability Maturity Model (CMM), which aims to improve software development processes to deliver high-quality software within budget and planned cycle time. Adopting structured approaches like CMM can result in improved software quality, reduced defects, and increased efficiency in the software development lifecycle [17].

Analyzing code quality across different programming languages and frameworks, such as Java, Kotlin, and Flutter, can offer valuable insights into the impact of development choices on software quality. Leveraging tools like SonarQube for evaluating and comparing code quality metrics can provide a quantitative basis for assessing the effectiveness of development practices and identifying areas for improvement. Developers can enhance software applications' overall quality and reliability by systematically analyzing code quality indicators and addressing issues such as code smells and complexity, improving user experiences and long-term success [18].

1.6 Research Gap

Despite the critical role of mobile applications in various sectors and the rapid evolution of development technologies, there remains a significant gap in empirical research regarding the comparative analysis of mobile development approaches, particularly regarding their impact on code quality. Most existing studies focus on individual aspects of development, such as usability, performance, or developer preferences. Still, few comprehensively evaluate how different programming languages and frameworks—like Java, Kotlin, and Flutter—affect code's overall quality and maintainability.

Current literature extensively discusses the features and benefits of Kotlin and Flutter, noting their potential to streamline the development process and enhance code safety and maintainability. However, more systematic empirical studies need to quantify the impact of these modern languages and frameworks on code quality in real-world development scenarios. This includes a detailed analysis of code smells, which are subtle indicators of potential future problems or technical debt that might not currently affect an application's functionality but could lead to significant maintenance challenges.

Furthermore, while tools like SonarQube offer capabilities to assess and compare code quality metrics across different environments, the practical application of these tools in comparative studies must be extensively documented in academic research. Studies that not only use these tools to gather data but also critically analyze this data to provide actionable insights into how specific characteristics of Java, Kotlin, and Flutter influence the maintainability, scalability, and efficiency of the development process are needed.

This research aims to fill these gaps by conducting a rigorous comparative analysis of mobile applications developed in Java, Kotlin, and Flutter. It will evaluate various code quality metrics, such as the prevalence and severity of code smells and the overall complexity of the codebases, to determine the tangible impacts of choosing one technology over another. This study will provide empirical evidence to guide developers in selecting the most suitable programming language or framework for their projects based on quantifiable code quality and maintainability measures.

By addressing this research gap, the study will contribute valuable insights to the field of software engineering, particularly mobile app development. It will enhance understanding of the practical implications of development tool choices on long-term application success and sustainability.

1.7 Study Objectives and Research Questions

The overarching objective of this thesis is to conduct a thorough empirical analysis to compare the impact of different mobile development approaches—specifically Java, Kotlin, and Flutter—on the quality and maintainability of mobile applications.

This research is driven by the need to provide developers and stakeholders with data-driven insights that can guide their decisions regarding which development technologies to adopt, depending on specific project requirements and goals.

The choice of focusing on these specific technologies is substantiated by their prevalent use and perceived benefits within the development community. According to a 2022 developer survey, Flutter has emerged as the most popular cross-platform mobile framework. The survey reports that 46 percent of software developers used Flutter, highlighting its significant adoption. This statistic is particularly compelling, considering that approximately one-third of mobile developers utilize cross-platform technologies, while the remainder prefer native tools.

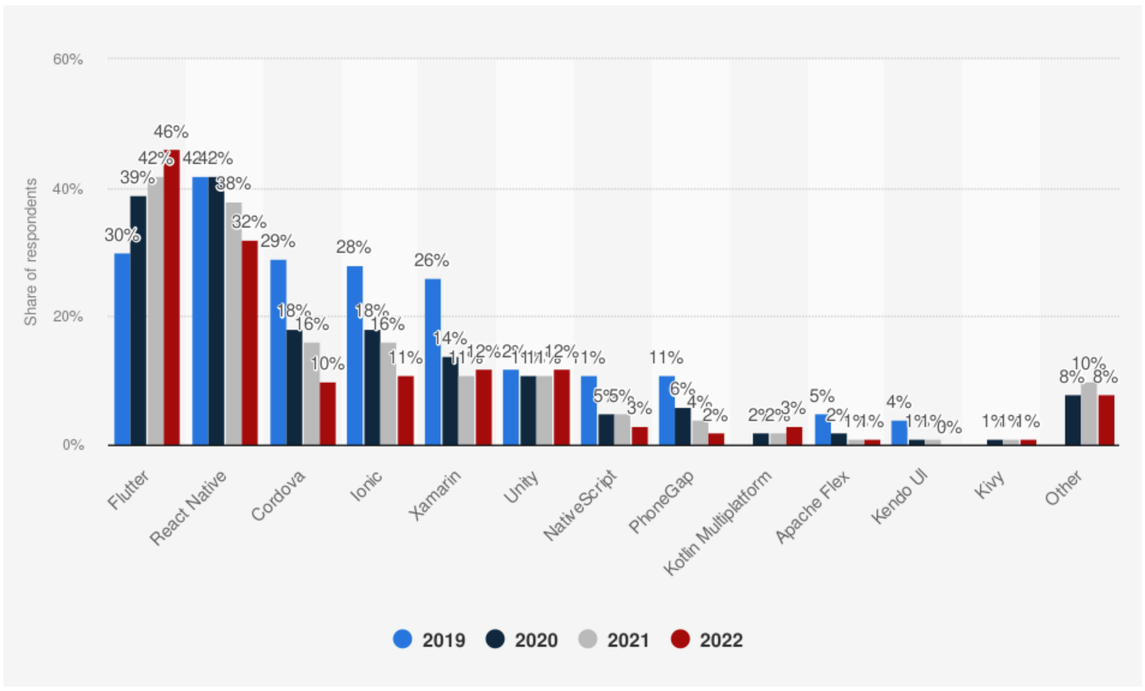


Figure 1.2: Cross-platform mobile frameworks used by software developers worldwide from 2019 to 2022 [2]

Objectives

1. To evaluate the development efficiency of Java, Kotlin, and Dart. This includes measuring the time taken for development, the ease of implementation of features, and the integration of third-party services across the three languages.

2. To analyze the maintainability of Java, Kotlin, and Dart applications. Maintainability will be assessed by examining code complexity, readability, and the ability to adapt or extend the application with new features.

3. This study aims to compare the code quality of applications developed using Java, Kotlin, and Dart(Flutter). Code quality will be evaluated based on the prevalence of code smells, adherence to coding standards, and the occurrence of bugs or issues during and after development.

4. To provide recommendations on the choice of development approach based on empirical data: Based on the findings, the study aims to offer guidelines on selecting development approaches for different mobile app projects, considering factors like project size, complexity, and specific industry needs.

Research Questions

In pursuit of these objectives, the study will seek to answer the following research questions:

1. Comparing the Efficiency and Resource Utilization of Java, Kotlin, and Dart: How does each technology stack up against one another regarding development efficiency, time, and resources required to create a fully operational mobile application? What are the implications of selecting each technology for the development process?

2. Considering the Impact on Maintenance and Long-Term Code Quality: What are the consequences of choosing Java, Kotlin, or Dart on mobile application maintain-

ability? How do code complexity, debugging ease, and change adaptability impact long-term maintenance and code quality in these development environments?

3. Navigating Coding Challenges and Selection Criteria: How does each development approach—Java, Kotlin, or Dart—address prevalent coding challenges and quality issues, such as the frequency and severity of code smells? Based on these considerations, which development approach is most desirable for various mobile application projects? Considering project scope, performance requirements, and target platforms, what are the recommendations for language selection to optimize development outcomes?

1.8 Methodology, Significance, and Structure of Thesis

A robust methodology has been adopted to achieve the research objectives outlined in this study, which involves developing a Kanban board application using three distinct programming environments: Java, Kotlin, and Flutter. The subsequent analysis of these applications will employ SonarQube, a sophisticated tool designed to assess metrics such as lines of code (LOC), code smells, and overall code quality. This method not only facilitates the collection of quantitative data concerning the efficiency and maintainability of each language but also provides qualitative insights into the coding experience. Such a dual-focused analysis enables a comprehensive understanding of the practical impacts of each development approach on project outcomes.

The insights derived from this research are expected to be highly valuable for developers and project managers, guiding the selection of development tools and practices based on empirical evidence. By clearly delineating the strengths and weaknesses associated with Java, Kotlin, and Flutter, this study contributes to informed decision-making within the software development community. Such knowledge is crucial for enhancing app quality and developer satisfaction, and it is anticipated that the findings will encourage more efficient and effective development practices within the industry. The practical implications of this research are significant, offering potential improvements in development processes, resource allocation, and project management in mobile app development.

This thesis is methodically structured into six comprehensive chapters, designed to guide the reader through the research process systematically:

- **Introduction:** This opening chapter sets the stage by outlining the motivation for the study, the research gaps identified, and the objectives to be achieved.
- **Literature Review:** This section delves into a thorough review of existing research, discussing the theoretical frameworks and previous empirical studies relevant to mobile application development, mainly focusing on Java, Kotlin, and Dart(Flutter).
- **Methodology:** This section details the specific methods employed in the study, including the development processes for the Kanban board application in each programming environment and the analytical tools and criteria used to evaluate code quality.
- **Results:** This paper presents a detailed account of the application development and analysis findings, offering a comparative perspective on the code quality metrics observed across the different programming environments.
- **Discussion:** This section interprets the results in the context of existing literature and discusses their implications for developers and the broader software development industry.
- **Conclusion and Recommendations:** This section summarizes the findings, discusses the study's potential limitations, and offers recommendations for future research and practical applications in mobile app development.

II Literature Review

Building upon the discussion of Kotlin's integration into Android development, it is crucial to contextualize this transition by considering the origins and foundational concepts of the fundamental programming languages and frameworks in use today. To this end, we shall delve into the historical development of Kotlin, Java, and Flutter. Each of these technologies has uniquely contributed to the evolution of software development practices and choices available to mobile developers. A brief exploration of their inception and initial objectives will provide valuable insights into their current roles and capabilities within the technology landscape.

2.1 Java

Java was developed in 1995 by Sun Microsystems, with the initial release being Java 1.0. It was designed by James Gosling and his team as a programming language that could run on any device without recompilation, known as "write once, run anywhere" [19]. Java's structure is based on classes and objects, following the object-oriented programming paradigm. It is a statically typed language, meaning variables must be declared before they can be used, enhancing code reliability and maintainability [20].

Java is widely used for Android app development due to its portability, security features, and extensive standard library. Android apps are primarily developed in Java, making it the de facto language for Android development [21]. The Android Software Development Kit (SDK) provides developers with tools to build apps efficiently, leveraging Java's robust features. Additionally, Java's platform independence allows Android apps to run on various devices with different hardware and software configurations, contributing to its popularity in the mobile app development industry [21].

In the context of Android, Java is utilized to access Android APIs, enabling developers to interact with the underlying operating system and create feature-rich applications [21]. The structure of Java facilitates the development of complex Android apps by providing a well-defined syntax, memory management through garbage collection, and support for multithreading, which is essential for responsive and interactive mobile applications [20].

2.2 Kotlin

Kotlin, a statically typed programming language supporting object-oriented and functional programming, was developed in 2011 by JetBrains. Inspired by various languages like Groovy, Kotlin aimed to address the limitations of existing languages and provide enhanced features for developers [22]. Kotlin's structure is designed to be concise and expressive, offering features like improved conditional execution with the "when" structure, which allows for more optimal handling of business tasks. Additionally, Kotlin incorporates modern language features such as null-safe navigation and coroutines for asynchronous programming [23].

Kotlin is widely used for Android app development due to its interoperability with Java, seamless integration with Android Studio, and concise syntax, which increase developer productivity [23]. The language's versatility and compatibility with existing Java codebases make it a popular choice for building Android applications. Furthermore, Kotlin's safety features, such as null safety and immutability by default, contribute to writing robust and bug-free code.

In Android development, Kotlin provides developers with tools like the Kotlin Coroutines library for asynchronous programming and the Gradle Kotlin DSL for project assembly, enhancing the development process [22]. Its concise syntax and modern features enable developers to create efficient and maintainable Android applications [24]. Kotlin's popularity in Android development is further evidenced by its use in various research projects focusing on Android-based applications.

2.3 Dart(Flutter)

Flutter is a versatile app SDK developed by Google that allows developers to create high-performance applications for various platforms like iOS, Android, and the web using a single codebase [25]. It was initially released in 2017 and has gained popularity due to its ability to provide a consistent user experience across different platforms.

Flutter's architecture is based on the Dart programming language, which Google also developed. Dart is an object-oriented, class-based language that supports interfaces, mixins, abstract classes, and optional typing. It is optimized for building user interfaces and allows for ahead-of-time compilation of native code for better performance [26].

Flutter uses a reactive framework composed of widgets, which are the building blocks of Flutter applications. Widgets in Flutter are arranged hierarchically to create the user interface. Flutter's architecture allows for hot reload, which enables developers to see the changes they make to the code reflected in the app almost instantly, making the development process more efficient [27].

One of the critical reasons why Flutter is used is its ability to create cross-platform applications with a single codebase. This significantly reduces development time and effort as developers do not have to write separate code for different platforms. Additionally, Flutter provides a rich set of customizable widgets, a fast development cycle, and strong community support, making it an attractive choice for developers [27].

2.4 Previous Research

Kotlin's Transition and Adoption in Android Development

The transition from Java to Kotlin in Android development is motivated by Kotlin's advanced features like null safety, extension functions, and concise syntax, which collectively enhance code readability and maintainability. Kotlin's interoperability with Java and official support by Google further drive its adoption despite challenges related to the learning curve and migration efforts [28]. On the other hand, Flutter's emergence as a cross-platform framework offers the advantage of natively compiled applications from a single codebase across various platforms, supported by its widget-based architecture and the Dart programming language [29]. However, concerns persist regarding Flutter's performance compared to native applications and the maturity of its ecosystem.

Recent research by V. Oliveira, L. Teixeira and F. Ebert. [30] explores the adoption of Kotlin for Android development through a mixed-methods approach, combining quantitative analysis of Stack Overflow discussions with qualitative interviews of Android developers. This study highlights the rapid acceptance of Kotlin following its endorsement by Google, attributing its popularity to features like null safety, expressiveness, and interoperability with Java. These attributes have facilitated a smoother transition for developers migrating from Java, enabling them to utilize existing Java libraries while benefiting from Kotlin's modern features.

Developers appreciate Kotlin's concise syntax and enhanced safety features [30], contributing to more robust and maintainable code. However, the study also reveals challenges, such as the complexity introduced by Kotlin's functional programming capabilities, which some developers find obscure and challenging to manage in larger codebases. Interoperability with Java, while largely beneficial, introduces its complexities, mainly when dealing with nullability and Java's legacy code.

To determine the effect of code smells on software maintainability, Matheus Flauzino conducted an empirical analysis of the prevalence of code smells in Java and Kotlin, examining more than 6 million lines of code from 100 GitHub repositories. This paper is essential for developers debating whether to use Java or Kotlin since it shows conclusively that the latter tends to have fewer code smells, which is a surefire sign of future maintenance issues. The study analyzed five prevalent code smells and found that Kotlin consistently displays fewer examples of these troublesome patterns except for the Long Parameter List than Java. The results imply that Kotlin's design, which prioritizes readability and conciseness, may naturally prevent code smells from developing. Because of this, Flauzino et al.'s paper highlights the complex factors in selecting a programming language for software development and presents a strong case for using Kotlin over Java for projects where maintainability is a concern [31].

Improving Code Quality and Software Maintenance

The focus on code quality in mobile app development, including integrating tools like SonarQube, emphasizes the importance of managing technical debt and addressing code smells [32]. Innovations like ecoCode emphasize energy-efficient coding practices, reflecting a broader concern for sustainable software development. Comparative studies on development methodologies provide insights into the challenges and considerations involved in transitioning between frameworks, offering valuable practical experiences for developer [33].

Evaluating open-source projects offers practical insights into applying different development methodologies, aiding in identifying patterns, best practices, and common pitfalls associated with various programming languages and frameworks. Despite the wealth of knowledge on mobile app development frameworks, there is a gap in research that combines practical development experience with a thorough analysis of open-source projects across diverse frameworks, particularly in exploring the effects of methodologies on code quality metrics like code smells and their severity [11].

The prediction of code smells in continuous integration is a critical task for quality managers and developers. A recent study by Md Abdullah Al Mamun [34] investigated the effectiveness of organic versus cumulative software metrics in predicting code smells. The researchers used over 36,000 software revisions from 242 open-source Java projects to develop predictive models. Their findings revealed that non-cumulative (organic) metrics, which reflect changes between revisions rather than aggregated totals, were significantly more effective in predicting code smells. This is because organic metrics are more sensitive to recent changes in code, offering a more accurate measure for predicting software quality issues.

In contrast, cumulative metrics were less effective, potentially masking recent developments that were more indicative of current software quality. The distinction between organic and cumulative metrics is essential for quality managers and developers in continuous integration environments, where quick iterations and timely quality assessments are crucial. The study also employed various model validation techniques to ensure the generalizability of the results.

Static Analysis Tools and Their Application

Static analysis tools have been recognized as valuable for improving software quality. However, a recent study by D. Marcilio [35] critically examined SonarQube, a leading static analysis tool, in real-world software projects across various organizational settings, including open-source communities and government institutions. The study revealed that despite the perceived effectiveness of static analysis tools in enforcing coding standards and reducing technical debt, the actual resolution rate of identified issues remains low, with an average resolution rate of only 13

The study conducted by Ifeanyi Rowland Onyenweaku [36] aimed to evaluate the effectiveness of static analysis tools in identifying defects in software applications, with a focus on the use of SonarQube in analyzing Spectral Workbench, a tool used for capturing and analyzing spectral data. The researchers found that SonarQube successfully identified many defects, ranging from minor UI issues to critical bugs, which could cause system failures if not addressed. Their analysis revealed 232 code smells and 63 bugs, providing a detailed categorization of issues by severity and type, which could aid developers in prioritizing which issues to address first. The study also highlighted the challenge of low-resolution rates for identified issues in software development, with constraints such as prioritization, resource allocation, and perceived severity often playing a role. Despite this, the researchers emphasized the potential of integrating SonarQube into open-source projects, providing a blueprint for other projects. Overall, the findings of this study demonstrate the value of static analysis tools in enhancing the reliability and effectiveness of software applications while highlighting the challenges that need to be addressed to improve the resolution rate of identified issues.

The research conducted by Sebastian Stiernborg on the application of SonarQube in the open-source project Spectral Workbench has been instrumental in providing a deeper understanding of the practical usage of this tool in a corporate setting. The study presents a comprehensive view of the operational challenges and benefits associated with implementing continuous inspection processes, thereby contributing to the body of knowledge in this domain. In his master thesis [37], Sebastian Stiernborg delves into the efficacy of implementing continuous inspection processes within software development teams, specifically focusing on SonarQube. The study was conducted in collaboration with Furhat Robotics' development team, where Stiernborg worked closely to introduce SonarQube to their existing development processes. The primary objective of the integration was to automate code reviews and improve code quality without disrupting the developers' workflow. The study presents a set of guidelines based on feedback from integrating continuous inspection tools, highlighting the advantages of such tools, including the early detection of defects and enhanced code quality. However, the study also identifies potential challenges, such as the complexity of integrating these tools within existing systems and the need for careful handling to avoid disruption and ensure developer buy-in.

Furthermore, Stiernborg's research emphasizes the need for further studies to explore the integration of different continuous inspection tools and features. One of the key findings from the research is the challenge of balancing the immediate benefits of continuous inspection with the initial resistance and learning curve associated with adopting new tools. The study details the specific implementation steps taken at Furhat Robotics, including the challenges of configuring and maintaining such tools within existing systems and the strategies employed to overcome these challenges. Overall, the study contributes to the growing body of research on continuous inspection processes and highlights the importance of careful planning, implementation, and evaluation of such tools in software development teams.

The study conducted by Sebastian Stiernborg [37] on integrating static analysis tools and its impacts on maintenance activities was further enriched by Ze phyrin Soh's [38] investigation. Soh's study quantified the specific effects of code smells on various maintenance tasks, offering a more detailed perspective on the implications of code quality on software maintainability. Soh's research is particularly noteworthy as it differentiates between different types of maintenance efforts, such as editing, navigating, reading, and searching, unlike previous studies that treated maintenance effort as a monolithic activity. The study revealed that code smells have varying impacts on different maintenance activities. While some code smells may increase the effort required for tasks such as navigating and editing, they do not uniformly affect all maintenance activities. For instance, the code smell "Feature Envy" significantly increased the effort required for searching activities, whereas "Data Clumps" primarily elevated the effort in editing tasks. This nuanced view provides a more granular understanding of developers' practical challenges when dealing with code smells in a maintenance context. The study also utilized a sophisticated methodological approach, utilizing multiple linear regressions to analyze the impact of code smells while considering other factors such as file size and number of revisions. This approach validates the significant role of code smells in increasing maintenance effort and refines our understanding of their impact relative to other factors such as file size and changes made.

Ensuring high-quality code is fundamental to developing effective and efficient software engineering software. However, recent research [39] conducted by Corral and Fronza challenges the notion that superior code quality is the primary determinant of app success in the Google Play store. The study examined the relationship between source code quality and market success indicators, such as the number of downloads and user ratings, using a sample of 100 open-source mobile applications. Results revealed a relatively marginal impact of source code quality on market success indicators, suggesting that factors such as marketing strategies, app functionality, user interface design, and overall user experience play more significant roles in determining the success of mobile applications in app stores. The methodology employed by Corral and Fronza involved using statistical methods to establish a strong correlation between established source code metrics and app store success metrics. The findings underscore the complexity of app success and highlight the limitations of relying solely on technical excellence in a market driven by consumer preferences and competitive features [39]. In the research conducted by Luis Corral and Ilenia Fronza, Figure 2.1 showcases a visual representation of the market success model based on app store metrics such as Number of Downloads (NOD), Number of Reviewers (NOR), and Application Rating (AR). This figure effectively demonstrates how these metrics collectively contribute to defining the market success of mobile applications. NOD measures the popularity and reach of an app by indicating how many times it has been downloaded, thus indirectly measuring its visibility and user interest. NOR, on the other hand, represents users' engagement with the app by counting how many users have taken the time to review it. Lastly, AR, which is averaged from all user ratings, offers a direct insight into user satisfaction and perceived app quality. Combined, these metrics provide a comprehensive view of an app's performance in the market. They portray how often it was downloaded, how well users received it, and the extent of active user interaction and satisfaction.

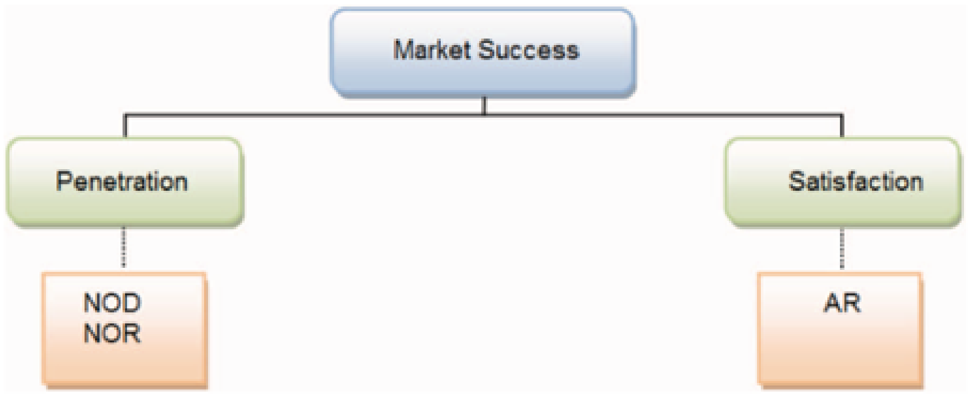


Figure 2.1: Corral and Fronza's research resulted in a Market Success Model Based on App Store Metrics.

The study by Michele Tufano [40] sheds light on the origins and persistence of code smells in significant software ecosystems, presenting critical insights into the internal software development processes. The empirical analysis conducted on 200 open-source projects from Android, Apache, and Eclipse ecosystems traces the history of code changes to pinpoint when specific code smells are introduced and under what conditions they are most likely to occur. The study [40] finds that many smells are introduced at the inception of code entities or through changes that do not directly relate to ongoing maintenance, challenging the prevailing assumption that code smells primarily emerge during routine maintenance and evolution activities. The research used a metric-based methodology to detect smell-induced changes, providing a granular view of how smells develop over time. The findings suggest that tool-based smell detection and refactoring recommendations should consider the specific developmental context to be truly effective and advocate for the development of more sophisticated recommendation systems that could help developers plan and implement refactoring activities more strategically, thereby potentially reducing the introduction of new smells during both development and maintenance phases.

Building on Michele Tufano's analysis [41] of the origins of code smells, the research [40] conducted by Tarek Alkhaeir and Bartosz Walter sheds further light on the significant impact these smells have on the relationship between design patterns and defects. Their study [40] highlights that code smells can significantly exacerbate the likelihood of defects in software projects where design patterns are implemented. Through an analysis of Java classes from ten systems in the PROMISE dataset, Alkhaeir and Walter critically evaluate how design patterns and code smells correlate with increased defects. Their work confirms that classes utilizing design patterns are not inherently defect-prone but become so when code smells are present. The study establishes a clear connection between smelly and non-smelly design pattern classes and defects using statistical tests. The findings indicate that pattern classes with code smells experience more defects than their non-smelly counterparts, underscoring the substantial negative impact of code smells on software quality. The researchers also observed that while non-smelly design patterns tend to have a neutral or slightly negative effect on defect proneness, introducing code smells drastically increases the likelihood of defects.

Xiaofeng Han and colleagues conducted a study [42] to explore the practical applications of code reviews in identifying and addressing code smells within the OpenStack community. Drawing on prior research on code smells in software systems, the study reveals that code smells are critical indicators of potential maintenance issues often overlooked during the review process. Specifically, the researchers examined code reviews within the Nova and Neutron projects and found that coding convention violations or oversights by developers were common causes of code smells. However, when code smells were identified, reviewers provided actionable recommendations for refactoring, typically implemented by developers, improving code quality. The study highlights the value of manual code review over automated tools, as the former is more context-sensitive and effective in identifying subtle issues. The researchers found that enhancing the code review process with better guidelines and training on smell detection could further improve the effectiveness of this quality assurance practice. This collaborative approach to code review reinforces the importance of adhering to coding standards and demonstrates that code reviews are crucial to identifying and addressing code smells despite the challenges involved.

Emergence and Impact of Flutter in Cross-Platform Development

Cross-platform development tools, including Flutter, are rapidly transforming the landscape of mobile application development by offering significant reductions in code complexity and enhancing maintainability—a study by [43]. Y. Cheon and C. Chavez demonstrated that when an existing Android application was rewritten in Flutter, it could run on Android and iOS platforms and required approximately 37% fewer lines of code than the original Java implementation [43]. This reduction is primarily attributed to Flutter's streamlined approach to UI development and robust widget library, which significantly diminishes the need for verbose UI code and extensive platform-specific adaptations.

Furthermore, the transition from Java to Flutter revealed that Flutter's declarative UI framework and reactive programming model could improve application performance and responsiveness. By moving away from the imperative and stateful approaches typical in Android development, Flutter enables more dynamic and efficient handling of UI changes, leading to smoother user experiences [44] This shift is crucial for developers looking to build highly interactive and responsive applications without the overhead of managing complex state synchronization across user interface components.

The triangulation method used in the study—analyzing discourse from Stack Overflow alongside direct developer interviews—provides a comprehensive understanding of the practical challenges and benefits developers observe in real-world scenarios. This approach confirms the theoretical advantages of Kotlin discussed in previous literature and illuminates the nuanced difficulties encountered during its practical application.

The evolution of programming languages like Java and Kotlin and frameworks such as Flutter has significantly shaped the landscape of Android application development. Each technology brings unique strengths for mobile app creation, from Java's "write once, run anywhere" principle to Kotlin's modern syntactic features and Flutter's cross-platform capabilities. This diversity allows developers to select technologies that best fit their project's requirements and personal or team proficiency. However, as the technology landscape continuously evolves, developers must remain adaptable, continually updating their skills and understanding of these tools. The ongoing transition towards more efficient, readable, and maintainable coding practices suggests a promising future for mobile development. By embracing these changes and learning from comparative and empirical studies, developers can better navigate the complexities of modern app development, ensuring robust, efficient, and user-friendly applications.

III Methodology

3.1 Overview

The following text outlines the comprehensive methodology adopted to conduct a comparative analysis of mobile application development approaches using Java, Kotlin, and Dart. The primary objective of this research is to evaluate and compare the development efficiency, code maintainability, and overall quality of applications developed across these three different programming languages. The methodology is divided into two distinct but interrelated parts to achieve a robust analysis.

Part I: Identical Mobile Application Development—This segment focuses on the practical aspect of developing an identical Kanban board application across the three selected programming languages. It involves setting up development environments, detailing the application features and implementation strategies, and standardizing development processes to ensure comparability. This part aims to directly assess the hands-on development experiences and the intrinsic differences in coding practices, development time, and initial code quality among Java, Kotlin, and Flutter.

Part II: SonarQube Code Inspection and Open Source Projects - The second part of the methodology utilizes SonarQube, a tool for measuring code quality, to conduct an extensive analysis of existing open-source projects written in Java, Kotlin, and Dart. This includes a detailed setup of the SonarQube environment, a selection of projects based on predefined criteria, and systematic code inspections. The goal is to objectively evaluate and compare code quality in terms of maintainability, prevalence of code smells, and other quality metrics across projects that use these technologies.

The dual perspective on software development practices provided by both parts, direct through application development and indirect through analysis of existing codebases, enriches the understanding of each languages's capabilities and limitations and enhances the reliability of the study by cross-verifying findings from practical development with empirical data from broader project analyses.

By combining these methodologies, the research aims to deliver insightful conclusions that can guide developers in selecting the most suitable programming language based on empirical evidence and practical experiences. The subsequent sections will detail each part of the methodology, explaining the processes, tools, and criteria used to ensure a thorough and unbiased evaluation of each development approach.

3.2 Identical Mobile Application Development

The present study outlines a methodology for investigating the efficacy of three programming languages, Java, Kotlin, and Dart, by developing an identical Kanban board application. The investigation aims to evaluate each language's development efficiency, usability, and initial code quality. The study provides a comprehensive account of the developmental process followed for each language, ensuring that the functional parity of the application is maintained while adhering to the conversational practices of each programming language.

3.2.1 Project Planning.

The initial phase of our research project involved identifying an appropriate application to develop to assess which language would be best suited for project development and maintenance. The application needed advanced features to achieve this objective while remaining relatively uncomplicated. Initially, an e-commerce application was proposed. However, after careful consideration, it was determined that the backend development required for such an application would be too time-consuming. Consequently, a To-do application was chosen to fulfill the research objectives of identifying a language best suited for application development.

In the next phase of project development, the features to include and the backend and database to use were meticulously evaluated. After thorough consideration, it was concluded that Firebase is the optimal choice for the backend. This decision was primarily driven by Firebase's significantly reduced coding effort, thanks to its extensive range of pre-built features that seamlessly integrate into the project. Moreover, Firebase offers two database options—Realtime and Firestore—and Firestore was selected for its superior performance and scalability.

Once the database was settled on, a user-centric approach to feature selection was embarked upon. The features that best serve the project's needs were identified. The following features were chosen:

- Kanban Board Visualization: A visually appealing layout representing tasks across stages—to-do, In Progress, and Completed.
- User Account Management includes user registration, login/logout, and the option to log in with Google.
- Task Management is adding new tasks, editing and modifying existing tasks, and moving tasks between different stages.
- Time Tracking: This stopwatch feature allows users to track the time spent on tasks in the `"In Progress"` stage and pause the timer by moving tasks back to the To-Do stage.
- Export and Share Functionality: Users can export their tasks as a CSV file and share them with others.
- Database Integration: The Firebase database is utilized to store and manage user data and tasks, enabling real-time updates and synchronization across all devices.

The next step was to create a detailed design for the application. After some consideration, the Figma application was chosen due to its widespread popularity in app and web development. Establishing a comprehensive feature plan and precise design before commencing the coding process is essential. With a clear objective in mind, the importance of avoiding improvisation was recognized, leading to the decision to prioritize the design phase before diving into coding.

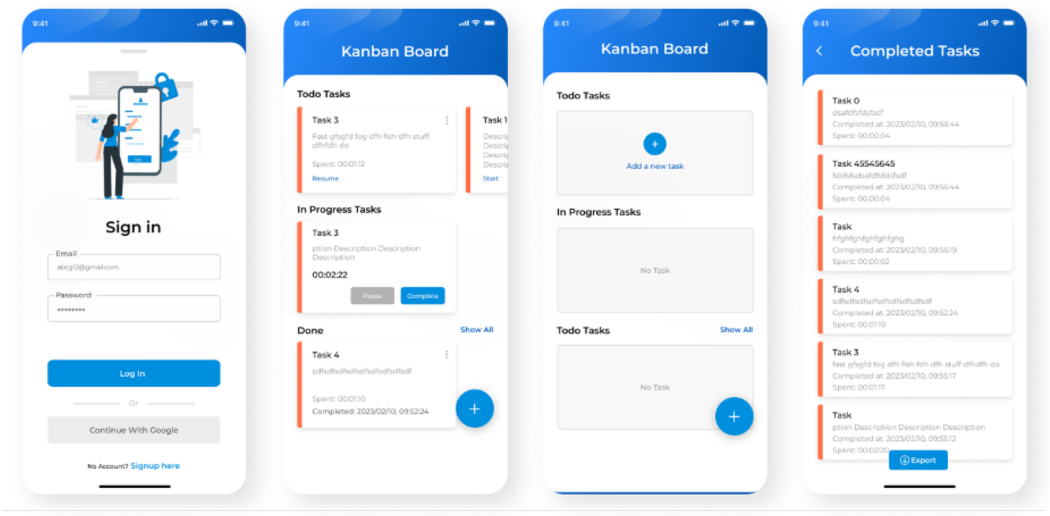
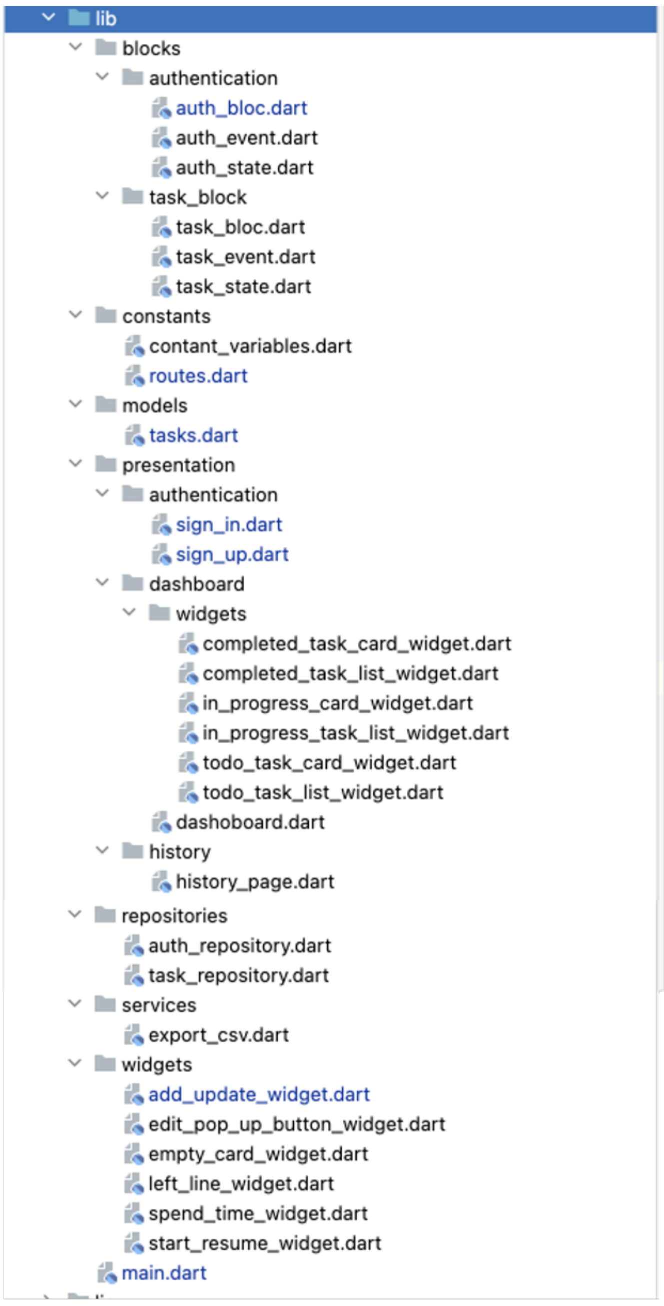


Figure 3.1: Planned view of Kanban Board Application. Designed Using Figma

Following the design phase of the project, during which Figma was utilized to create a comprehensive plan for the application, critical decisions regarding feature selection and the choice of database and backend service were made. With these decisions finalized, the project was poised to embark on the development phase, beginning with the coding part.

3.2.2 Flutter Application Development

The Flutter application was chosen as the starting point to ensure a smooth development process due to familiarity with the language. With feature specifications and Figma designs already in place, the process was streamlined. BLOC [45] was chosen for state management, which is highly regarded in the Flutter community for its reliability and ease of maintenance. Although it may require some practice and digging to get started, compared to other state management approaches like GetX [46] and Provider [47], BLOC is known for its reliability and ease of maintaining advantages. It is used solely for the business logic of the application and state management, separating it from the presentation layer. The three main components of BLOC are Events, States, and



```
lib
  blocks
    authentication
      auth_bloc.dart
      auth_event.dart
      auth_state.dart
    task_block
      task_bloc.dart
      task_event.dart
      task_state.dart
  constants
    contant_variables.dart
    routes.dart
  models
    tasks.dart
  presentation
    authentication
      sign_in.dart
      sign_up.dart
    dashboard
      widgets
        completed_task_card_widget.dart
        completed_task_list_widget.dart
        in_progress_card_widget.dart
        in_progress_task_list_widget.dart
        todo_task_card_widget.dart
        todo_task_list_widget.dart
      dashoboard.dart
    history
      history_page.dart
  repositories
    auth_repository.dart
    task_repository.dart
  services
    export_csv.dart
  widgets
    add_update_widget.dart
    edit_pop_up_button_widget.dart
    empty_card_widget.dart
    left_line_widget.dart
    spend_time_widget.dart
    start_resume_widget.dart
  main.dart
```

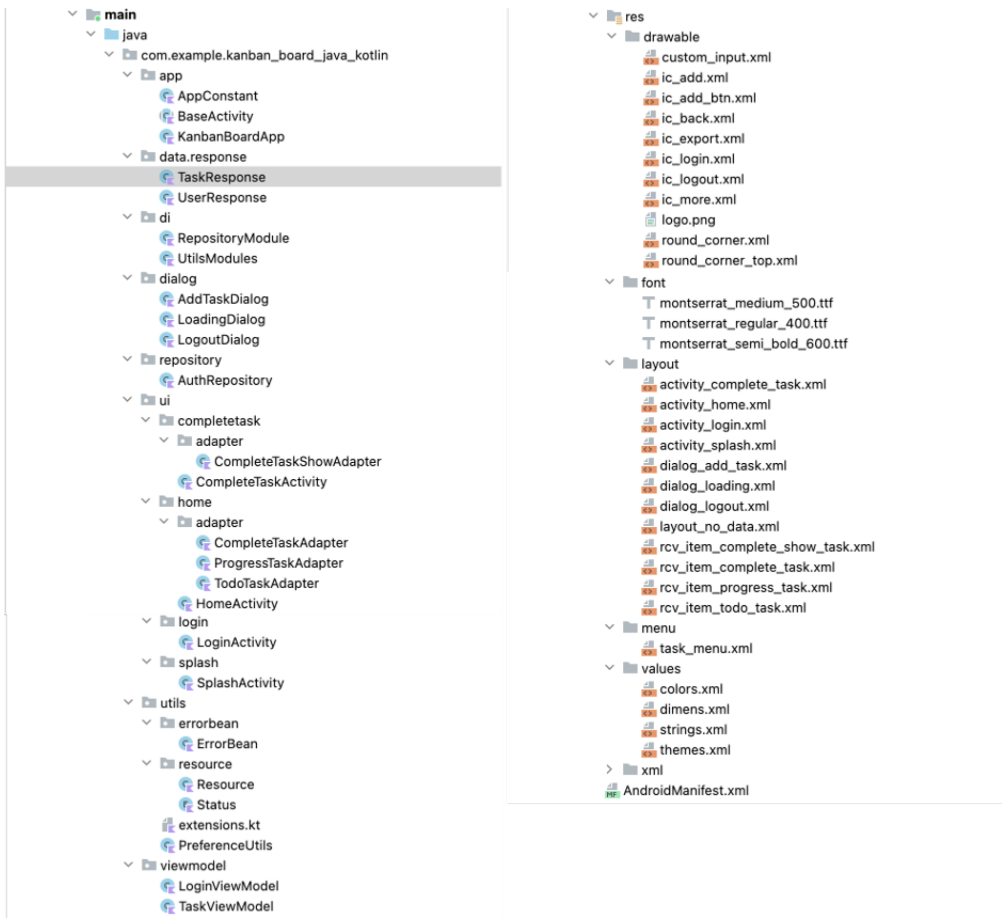
Figure 3.2: Flutter version of the application's project structuring

Blocs. User interactions trigger Events sent to a Bloc, which processes them and performs any necessary logic, such as data fetching or computations. The Bloc then emits a new State based on the results, and the UI listens to these state changes to update itself accordingly. This architecture enhances maintainability and testability by decoupling the UI from the core business logic.

The project took approximately 20 hours to complete, resulting in a total of 29 files. As the Figure 3.2 shows, the project was divided into directories. The project is organized into several directories: blocks contain BLoC components for authentication and task management, managing logic, events, and states; constants directory includes global constants and routing configurations; models define the task data model; presentation houses UI screens for authentication, a dashboard with various widgets for displaying tasks based on their status, and a history page for viewing completed tasks or activity logs; repositories manage data operations related to authentication and tasks, interfacing with databases or APIs; services includes functionalities such as exporting task data to CSV files; and widgets provides general-purpose UI components used across the application. The `"main.dart"` file acts as the app's entry point, setting up the environment and the root widget tree. The project structure suggests focusing on modularity and clean architecture to facilitate maintenance and scalability. While working on this project, the clean architecture principles mentioned in the book [48] by Robert Cecil Martin (also known as Uncle Bob) were followed.

3.2.3 Kotlin Application Development

After completing the initial Flutter project, a Kotlin-based Kanban board application was developed. Kotlin was adopted as the first choice for Android Native development. The Kotlin project, designed to support a Kanban board application, consisted of 26 *.kt files and 30 *.xml files. The development process took approximately 24 hours, primarily due to limited familiarity with Kotlin application development. However, despite these challenges, the Model-View-ViewModel (MVVM) architecture was leveraged to enhance maintainability and facilitate a clear separation of concerns. The MVVM [49] architecture enabled to segregate user interface logic from business logic, with ViewModel managing the UI-related data that can persist through configuration changes. At the same time, View handled the layout and display, and Model managed the data and business logic.



```
main
  java
    com.example.kanban_board_java_kotlin
      app
        AppConstant
        BaseActivity
        KanbanBoardApp
      data.response
        TaskResponse
        UserResponse
      di
        RepositoryModule
        UtilsModules
      dialog
        AddTaskDialog
        LoadingDialog
        LogoutDialog
      repository
        AuthRepository
      ui
        completetask
          adapter
            CompleteTaskShowAdapter
          CompleteTaskActivity
        home
          adapter
            CompleteTaskAdapter
            ProgressTaskAdapter
            TodoTaskAdapter
          HomeActivity
        login
          LoginActivity
        splash
          SplashActivity
      utils
        errorbean
          ErrorBean
        resource
          Resource
          Status
        extensions.kt
        PreferenceUtils
      viewmodel
        LoginViewModel
        TaskViewModel
```

```
res
  drawable
    custom_input.xml
    ic_add.xml
    ic_add_btn.xml
    ic_back.xml
    ic_export.xml
    ic_login.xml
    ic_logout.xml
    ic_more.xml
    logo.png
    round_corner.xml
    round_corner_top.xml
  font
    montserrat_medium_500.ttf
    montserrat_regular_400.ttf
    montserrat_semi_bold_600.ttf
  layout
    activity_complete_task.xml
    activity_home.xml
    activity_login.xml
    activity_splash.xml
    dialog_add_task.xml
    dialog_loading.xml
    dialog_logout.xml
    layout_no_data.xml
    rcv_item_complete_show_task.xml
    rcv_item_complete_task.xml
    rcv_item_progress_task.xml
    rcv_item_todo_task.xml
  menu
    task_menu.xml
  values
    colors.xml
    dimens.xml
    strings.xml
    themes.xml
  xml
AndroidManifest.xml
```

Figure 3.3: Kotlin version of the application's project structuring

As in Figure 3.3, the Kotlin code was orginized into intuitive packages: app for foundational classes like AppConstant and BaseActivity, data.response for handling data models like TaskResponse and UserResponse, and repository for data management, particularly with an AuthRepository to interface with authentication mechanisms. User interfaces were built under the "`ui`" package, divided into sub-packages like "`completetask`", "`home`", and "`login`", each containing activities and adapters for respective functionalities. Utility classes were stored in utils, with error handling and resource management centralized for accessibility.

The resource directory was meticulously structured with drawable resources for UI elements, layout XML files for defining user interfaces, and values for managing themes, strings, and dimensions, ensuring a consistent and visually appealing design. The AndroidManifest.xml was crucial in determining the application's configuration and permissions.

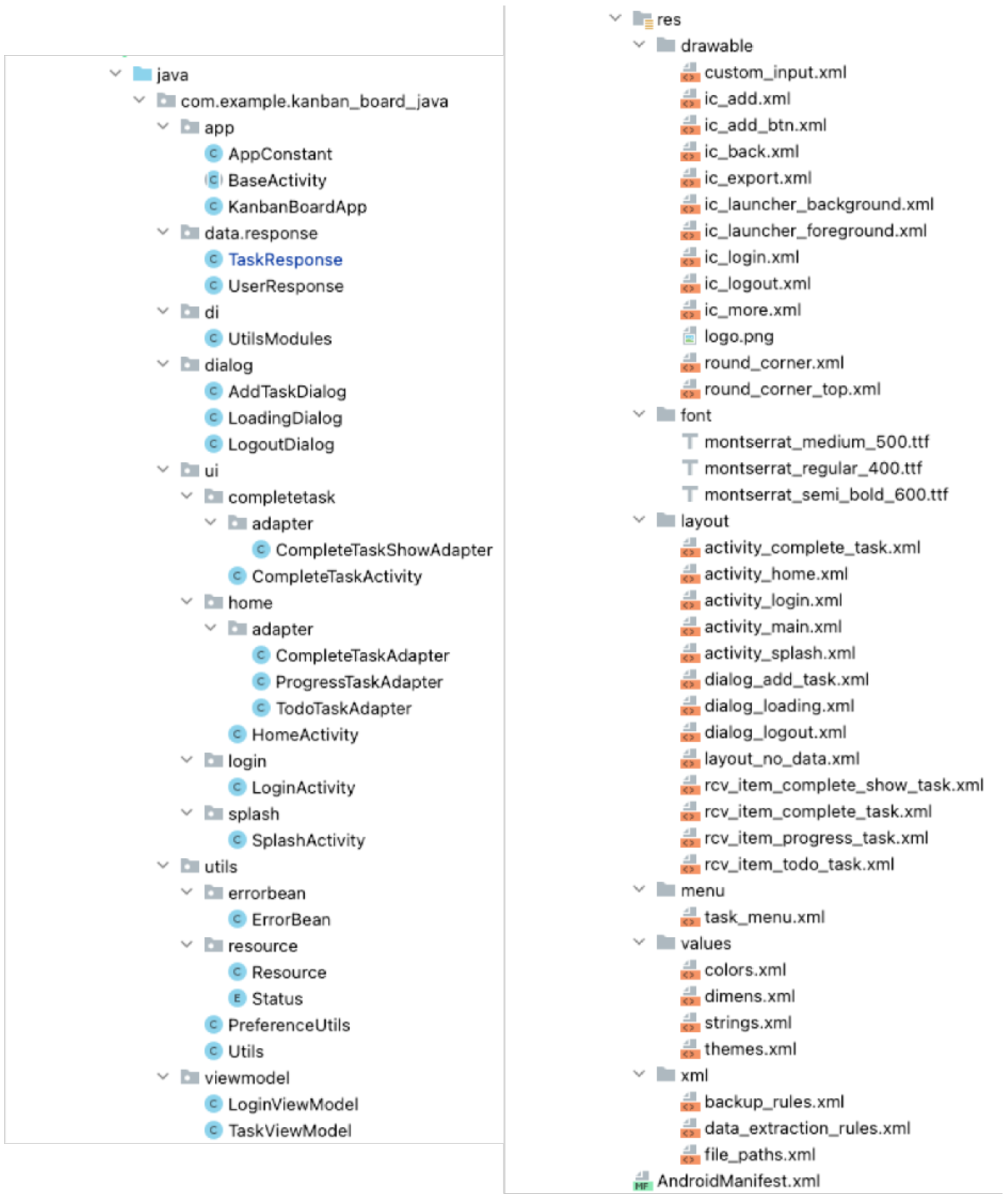
This Kotlin project, despite its inherent challenges, demonstrated the robustness of MVVM in Android development. It enabled a clean separation of business logic from the front end, improving the application's testability and scalability. Our design approach was heavily influenced by the clean architecture principles outlined in Robert C. Martin's works [48], focusing on creating a scalable, maintainable structure that could efficiently accommodate future enhancements and changes.

3.2.4 Java Application Development

Upon completing the Kotlin project, focus shifted to developing the Android application's Java version. To maintain consistency with the Kotlin version, advanced decompiling tools were implemented to facilitate the conversion of Kotlin code into Java. This approach significantly streamlined the development process, as the tools automatically translated the Kotlin code into Java, maintaining a similar project structure and functionality as outlined in the Kotlin project.

Compile tools proved remarkably effective given the tight project timeline and the need to expedite development without sacrificing code quality. By directly converting the Kotlin code to Java, considerable development time and effort were saved, allowing a focus on refining the Java application and ensuring its robustness and reliability. The entire process of translating and adjusting the Java project took approximately four hours.

The Figure 3.4 indicates that the resultant Java project was structured similarly



```
java
  com.example.kanban_board_java
    app
      AppConstant
      BaseActivity
      KanbanBoardApp
    data.response
      TaskResponse
      UserResponse
    di
      UtilsModules
    dialog
      AddTaskDialog
      LoadingDialog
      LogoutDialog
    ui
      completetask
        adapter
          CompleteTaskShowAdapter
        CompleteTaskActivity
      home
        adapter
          CompleteTaskAdapter
          ProgressTaskAdapter
          TodoTaskAdapter
        HomeActivity
      login
        LoginActivity
      splash
        SplashActivity
    utils
      errorbean
        ErrorBean
      resource
        Resource
        Status
      PreferenceUtils
      Utils
    viewmodel
      LoginViewModel
      TaskViewModel
```

```
res
  drawable
    custom_input.xml
    ic_add.xml
    ic_add_btn.xml
    ic_back.xml
    ic_export.xml
    ic_launcher_background.xml
    ic_launcher_foreground.xml
    ic_login.xml
    ic_logout.xml
    ic_more.xml
    logo.png
    round_corner.xml
    round_corner_top.xml
  font
    montserrat_medium_500.ttf
    montserrat_regular_400.ttf
    montserrat_semi_bold_600.ttf
  layout
    activity_complete_task.xml
    activity_home.xml
    activity_login.xml
    activity_main.xml
    activity_splash.xml
    dialog_add_task.xml
    dialog_loading.xml
    dialog_logout.xml
    layout_no_data.xml
    rcv_item_complete_show_task.xml
    rcv_item_complete_task.xml
    rcv_item_progress_task.xml
    rcv_item_todo_task.xml
  menu
    task_menu.xml
  values
    colors.xml
    dimens.xml
    strings.xml
    themes.xml
  xml
    backup_rules.xml
    data_extraction_rules.xml
    file_paths.xml
AndroidManifest.xml
```

Figure 3.4: Java version of the application's project structuring

to the Kotlin version. It included the same packages, such as "`app`" for core application setups such as AppConstant and BaseActivity, "`data.response`" for data models (TaskResponse, UserResponse), and "`repository`" for handling data operations (AuthRepository). The user interface was organized under the "`ui`" package with subpackages for different functionalities, mirroring the Kotlin setup. Furthermore, utility classes were organized under "`utils,`" and resources were meticulously arranged as in the Kotlin project.

Compression tools ensured consistency between the two projects and enhanced maintainability and scalability, thanks to the established structure and the robustness of the MVVM architecture used in both the Kotlin and Java projects. In conclusion, the strategic use of decompile tools proved essential in meeting our project deadlines efficiently while maintaining high code quality standards and application performance.

3.3 SonarQube Code Inspection and Open Source Projects

To commence our discourse, let us delve into a comprehensive comprehension of SonarQube and its purpose. SonarQube is an open-source platform widely utilized for continuous code quality inspection through static analysis. It automatically reviews code to identify bugs, code smells, and security vulnerabilities across various programming languages [28]. It is a popular tool in academia and industry for managing technical debt, with a significant user base exceeding 120,000 worldwide [50]. SonarQube tracks and manages technical debt by monitoring issue fixes and code quality improvements over multiple system revisions [51]. Additionally, it calculates various metrics like lines of code and code complexity, ensuring code compliance with specific coding rules for common programming languages [52].

SonarQube's effectiveness extends to cognitive complexity reduction, as it integrates metrics positively correlated with source code understandability, making it a valuable tool for developers [53]. The platform allows for detecting violations, known as issues, aiding in predicting changes through coding rule violations [54]. Furthermore, SonarQube is instrumental in identifying software metrics and technical debt through static analysis, consolidating the functionalities of tools like Checkstyle and PMD [55].

3.3.1 Issue Severity in SonarQube

Within the scope of our study, considerable emphasis has been placed on the severity levels of issues discovered through SonarQube. These levels play a pivotal role in determining the prioritization of code corrections and comprehending the possible implications on application security and performance. SonarQube classifies each identified issue into one of its five severity levels, offering developers a guiding language to address the most critical problems as a priority.

- **BLOCKER:** Represents bugs with a high probability of impacting the application's behavior in production. Examples include memory leaks or unclosed JDBC connections. These are critical and require immediate attention and resolution.
- **CRITICAL:** Issues that might have a lower probability of impacting the application's production behavior or represent security flaws, such as an empty catch block or SQL injection vulnerabilities. These issues necessitate a prompt review.
- **MAJOR:** Quality flaws that could significantly impact developer productivity, such as uncovered code, duplicated blocks, or unused parameters. While these may not immediately affect application functionality, they impede efficient code management and scalability.
- **MINOR:** Minor quality flaws that may slightly affect developer productivity, like overly long lines or "switch" statements with fewer than three cases. These are less critical but still warrant consideration for code quality improvement.
- **INFO:** Issues that are neither bugs nor quality flaws, but rather findings that may be useful for code optimization or future reference without requiring immediate action.

3.3.2 Setting Up SonarQube Environment

Measures were taken to ensure robust code quality analysis for the project by setting up a SonarQube environment. The first step involved installing Docker on the local machine, which was necessary to run the SonarQube server. After completing the installation, the SonarQube setup was initiated.

Using the terminal on a Mac, the SonarQube Docker [56] image was pulled by executing the command:

```
$ docker pull sonarqube: 9.9.4.87374-community
```

Once completed, it was confirmed that no existing Docker containers were running using the command:

```
$ docker ps -a
```

before installing new instance. With the command `docker run -d | name sonarqube -p 9000:9000 sonarqube:9.9.4.87374-community` SonarQube server was started. A subsequent `docker ps -a` check confirmed that SonarQube was active and running without issues.

To verify the setup, SonarQube was accessed at the local URL http://localhost:9000/ and logged in using the default credentials: Username `admin` and Password `admin`. Seeing that SonarQube was correctly set up and operational with this step.

Since SonarQube does not support Dart natively, the programming language used for the Flutter project, a plugin called sonar-flutter [57] was installed, developed by the Flutter community. This plugin allowed Dart code analysis to be integrated into the SonarQube environment, enabling the maintenance of high code quality standards and consistency across development efforts. This integration proved essential for leveraging SonarQube's capabilities to meet the project's needs.

3.3.3 Applying SonarQube Analysis to the Kanban Board Project

The integration of SonarQube into the development process of the Kanban board application was instrumental in ensuring high code quality across three different programming languages: Kotlin, Java, and Dart. This section details the steps taken for each language to set up and utilize SonarQube for continuous inspection of code quality.

Kotlin-Based Kanban Board Implementation:

1. Setting Up SonarQube: Docker was used to host SonarQube, ensuring a consistent environment. The SonarQube server was accessible at http://localhost:9000 after initiating the Docker container with the command: `docker run -d | name sonarqube -p 9000:9000 sonarqube: 9.9.4.87374-community`

2. Project Configuration: In the Kotlin project, the SonarQube Gradle plugin was added to the build.gradle file (apply plugin: `"org.sonarqube"`), and necessary configurations were set within the sonarqube block to specify the project key, name, sources, and exclusions.

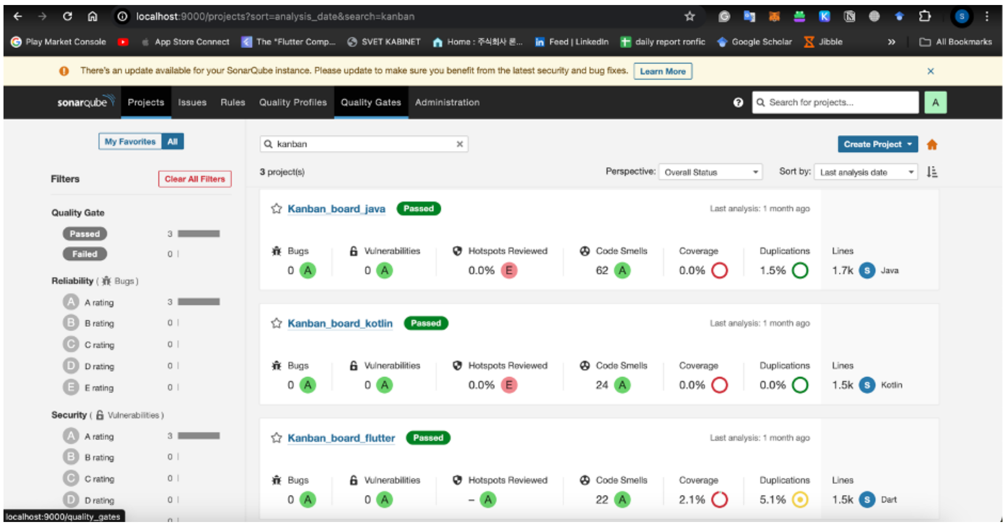
3. Running the Analysis: The analysis was triggered via the command `./gradlew sonarqube -Dsonar.host.url=http://localhost:9000 -Dsonar.login=$TOKEN` sending the code metrics to SonarQube and generating a detailed report on code quality.

Java-Based Kanban Board Implementation:

1. SonarQube Installation and Configuration: Similar steps were followed in the Kotlin implementation. Docker was used to deploy SonarQube, and the Java project was configured with the SonarQube plugin in the build.gradle file.

2. Analysis Configuration: Detailed properties were set in the sonarqube block of the build.gradle file, focusing on Java-specific paths and setting appropriate exclusions to filter out generated and test code.

3. Executing SonarQube Analysis: The SonarQube analysis for the Java project was executed using the same Gradle command. It analyzed the code quality and identified potential issues.



3 project(s)

Project	Quality Gate	Last analysis	Bugs	Vulnerabilities	Hotspots Reviewed	Code Smells	Coverage	Duplications	Lines
Kanban_board_java	Passed	1 month ago	0 A	0 A	0.0% E	62 A	0.0%	1.5%	1.7k Java
Kanban_board_kotlin	Passed	1 month ago	0 A	0 A	0.0% E	24 A	0.0%	0.0%	1.5k Kotlin
Kanban_board_flutter	Passed	1 month ago	0 A	0 A	– A	22 A	2.1%	5.1%	1.5k Dart

Figure 3.5: SonarQube Dashboard Overview: Code Quality Metrics for Kanban Board Projects in Flutter, Kotlin, and Java

Flutter-Based Kanban Board Implementation:

1. Plugin Installation for Dart Support: Since SonarQube does not officially support Dart, a third-party plugin (sonar-flutter-plugin-0.4.0.jar) was downloaded and added to the Docker container hosting SonarQube to enable Dart analysis.

2. Project Configuration for Flutter: In the Flutter project directory, a sonar-project.properties file was created, specifying the project key, name, and SonarQube token. This file also defined the source directories and set exclusions for files that should not be analyzed.

3. Running SonarQube Analysis: The Flutter project analysis was initiated with the command sonar-scanner after setting the necessary configurations in the properties file and running tests to generate coverage data.

3.3.4 Setting Up an Automated Pipeline for SonarQube Analysis of Open Source Projects

In the initial stages of the project, the inclusion of open-source project analysis was not planned. However, after implementing SonarQube into the Kanban board application, it was realized that deriving a statement based on a single project code written by one developer would not be sufficient to claim that one programming language or framework is better. To address this issue, SonarQube was implemented into GitHub open-source projects. A total of 42 open-source projects were selected from GitHub based on the number of forks and stars. With the Kanban board app developed during this research using three languages, the number of projects for analysis was 15 per language (14 open-source and 1 project developed internally). Tables with 15 projects in 3 languages are included in Appendix I.

However, a significant challenge was encountered due to the lengthy process of implementing SonarQube into the Kanban board application. It was noticed that creating a project on the SonarQube Dashboard and configuring data per language would be too time-consuming. Therefore, the intention was to automate processes such as downloading a GitHub project, creating a project in SonarQube, generating a token for the downloaded project, and saving tokens and other project configurations on the downloaded open-source projects. The first step in the automation process was learning about the APIs provided by SonarQube [58]. The Python script used for this automation is included in Appendix II, along with a detailed explanation.

In brief, the script downloaded open-source projects from GitHub, created a project with the same name on SonarQube, generated a token for that project, and saved all credentials in a text file for Java/Kotlin projects and a .yaml file for Flutter projects. The automation process helped save time, allowing more focus on project structuring rather than performing repetitive tasks.

After finishing the project preparation, each project still needed to be opened and configured. Once the configuration was completed as described earlier, reports were received within SonarQube. Another API provided by SonarQube was used, and another Python script was programmed, which is included in Appendix III. This script was one of the final steps of the development, helping to obtain the required statistics for the research.

IV Results of Comparative Analysis

4.1 Overview

The following chapter presents the findings of a comparative analysis of mobile application development using Java, Kotlin, and Dart (Flutter). The study primarily focuses on the SonarQube code quality assessments of identical Kanban board applications developed in each language. The severity of code smells and other metrics, such as the model of Task implementation in all three languages, were analyzed in detail. Accompanying tables and severity graphs provide a quantified representation of these findings. Additionally, the chapter offers insights into the practical differences between the languages, considering the application size and pertinent notes on the development process.

Further, the study delves into the implementation of SonarQube across a selection of open-source projects developed in Java, Kotlin, and Dart. It discusses the code quality metrics from these projects. The chapter includes graphs that provide detailed results across various categories. The final part of the chapter synthesizes the findings into a comprehensive graph highlighting the differences in code quality metrics between open-source projects across the three languages. The analysis presented in this chapter offers a broad perspective on the generalizability of the initial findings.

4.2 Results from Identical Application Development

Investigating the development of a Kanban board application is an essential part of this research that provides insightful data on the coding efficiency, maintainability, and overall quality of the code produced under different programming languages. In this study, Java, Kotlin, and Dart (Flutter) were utilized to generate identical applications, and the resultant findings provide valuable insights into each programming environment's inherent strengths and weaknesses.

4.2.1 Code Quality Analysis

The results of a detailed SonarQube analysis have brought to light significant disparities in the quality and efficiency of code amongst the three languages under scrutiny. As delineated in Table 4.1, Kotlin and Dart have demonstrated superior efficiency in coding practices compared to Java. Specifically, Kotlin has been found to require 1,467 lines of code (LOC), while Dart has slightly more, at 1,515 LOC. In contrast, Java has the highest LOC count at 1,748, indicating a higher level of complexity and potentially lower maintainability in its implementation.

Table 4.1: Severity levels of code smells detected in each language.

	Blocker	Critical	Major	Minor	Info	Total	LOC
Java	0	16	11	35	0	62	1748
Kotlin	0	7	3	14	0	24	1467
Dart	0	0	4	18	0	22	1515

Moreover, regarding code smells, Kotlin and Dart have exhibited a similar count of 24 and 22, respectively, signifying a comparable level of code quality between the two. Notably, Dart's codebase has shown no critical issues, implying robustness in security and stability. In contrast, Java's codebase has displayed many code smells, with 16 critical, 22 major, and 35 minor issues, indicating significant quality concerns and the need for more stringent quality control measures.

4.2.2 Task Model Implementation

The Task model's implementation, detailed in Appendix IV, showcases the differences in LOC required for each language—60 in Java, 17 in Kotlin, and 21 in Dart. Java's higher LOC can be attributed to the need for explicit getter and setter methods, which are not required in Kotlin due to its support for properties. Dart's LOC is less verbose due to its concise syntax. This disparity not only affects the amount of code written but also impacts the clarity and maintainability of the codebase.

4.2.3 Application Size and Performance

The compiled application APKs for both Java and Kotlin were found to be 11.5 MB in size, while the Flutter application size was 21.2 MB. This size difference can be attributed to the Flutter engine, which is essential for the cross-platform capabilities of Flutter applications. However, the performance testing across the three platforms showed minimal differences, suggesting that the Flutter application's increased size does not adversely affect its performance.

4.3 SonarQube Implementation in Open Source Projects

This section presents an in-depth analysis of the results obtained from the SonarQube analysis of various open-source projects developed using Java, Kotlin, and Dart (Flutter). A comparative examination is then conducted to identify and highlight the differences in code quality metrics across these languages. The findings reveal each languages's specific performance in handling code smells, which are presented with accompanying graphs to directly compare their efficacy in maintaining code quality in larger codebases. This academic approach enables readers to understand better the nuances and intricacies associated with each language's code quality and performance.

4.3.1 Java Projects Analysis

Java projects demonstrated a total Lines of Code (LOC) of 52,901, with 3,590 code smells identified. The distribution of these smells indicates a predominance of Minor code smells, accounting for 65%, as depicted in Figure 4.1 and Figure 4.4. Major issues comprised 23.3% of the smells, pointing to significant areas that could be optimized for better application stability and security. Critical issues were 9.5%, suggesting crucial vulnerabilities that need immediate attention. Informational and blocker issues were relatively low, at 1.6% and 0.7%, respectively.

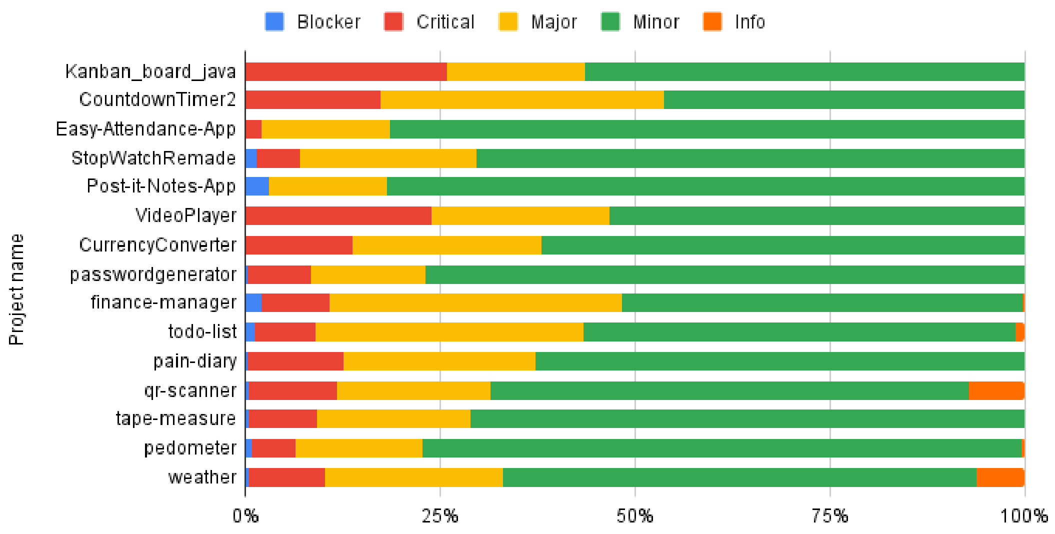


Figure 4.1: Severity Distribution in Java Projects

4.3.2 Kotlin Project Analysis

Kotlin projects showed a slightly lower total LOC at 48,865 but had a significantly lower total of 647 code smells. This reflects a more efficient codebase overall. However, the proportion of Critical issues was higher at 18.9%, as shown in Figure 4.2 and Figure 4.4, indicating that while the codebases are generally efficient, there are critical areas requiring mitigation to avoid potential vulnerabilities. Minor issues were 50.2%, major issues were 27%, and info issues were 3.9%, with no blocker issues recorded.

4.3.3 Flutter Projects Analysis

Flutter projects, using Dart for business logic and UI instead of XML as in Kotlin/-Java, exhibited a significantly larger total LOC at 174,044 but managed a relatively modest total of 1,303 code smells. The distribution favors Minor code smells at 58.8% and Major issues at 34.8%, underscoring Flutter's ability to manage larger codebases effectively. Critical issues were notably lower at 6.1%, and Info issues were minimal at 0.2%, as illustrated in Figure 4.3 and Figure 4.4. This distribution highlights Flutter's

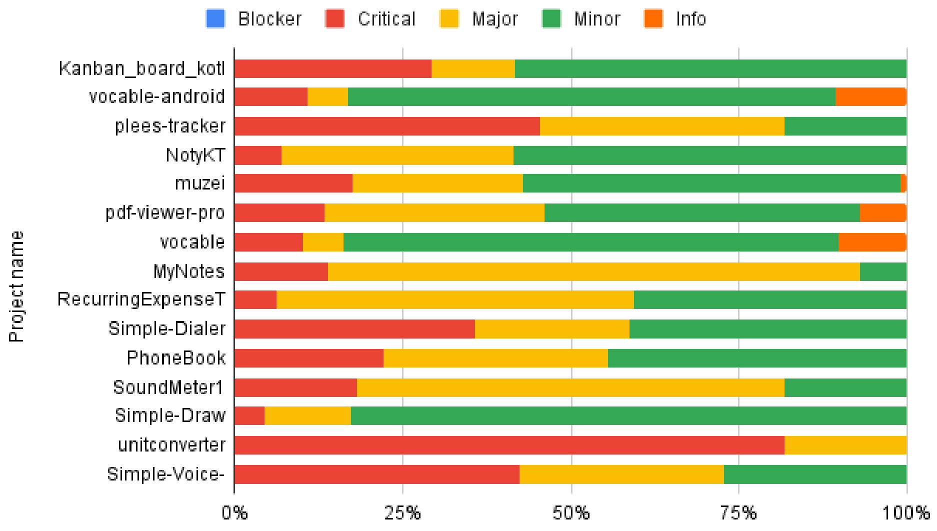


Figure 4.2: Severity Distribution in Kotlin Projects

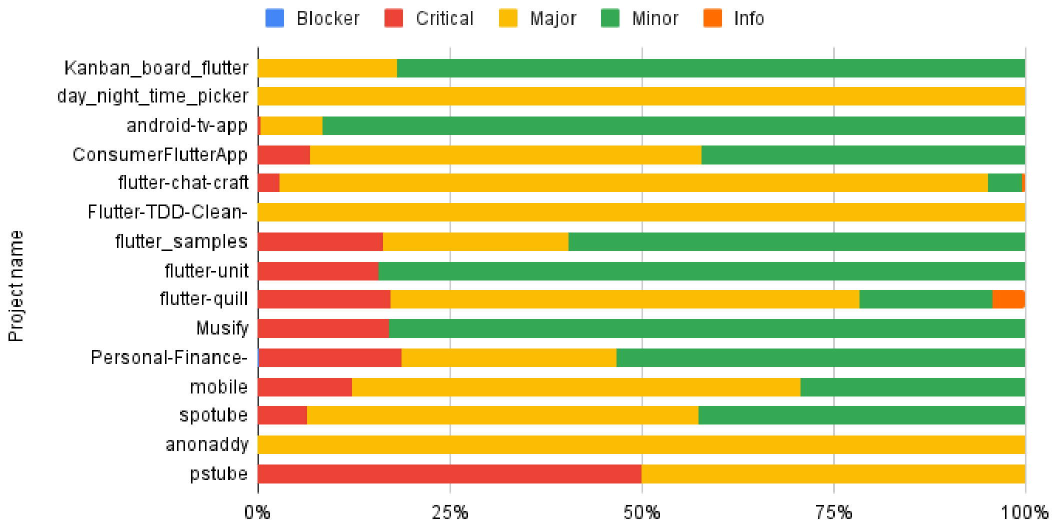


Figure 4.3: Severity Distribution in Dart Projects

capability to handle complex application development with fewer critical issues.

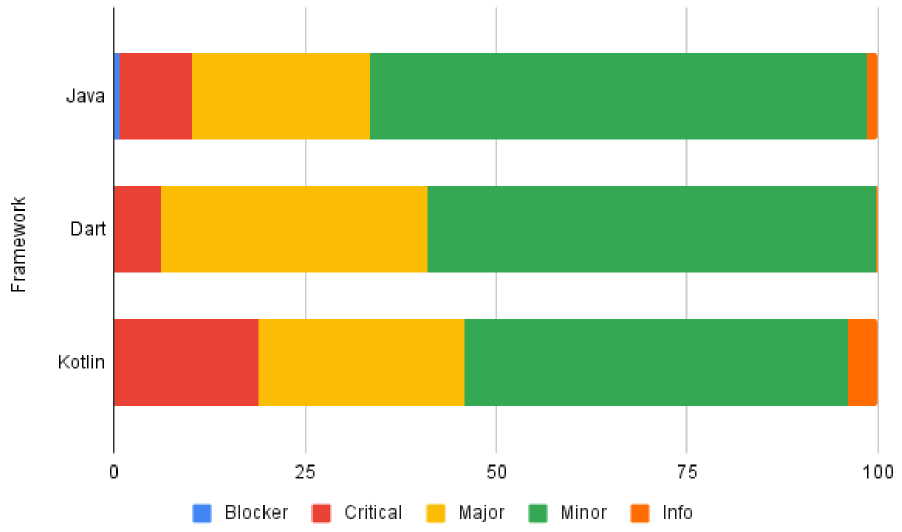


Figure 4.4: Severity Distribution of Code Smells Across Languages

4.3.4 Comparative Graph and Discussion

The comparative analysis, depicted in Figure 4.4, combines the data from Java, Kotlin, and Flutter projects, clearly visualizing how each language performs in managing code quality across different severity levels. This graph allows for a direct comparison and offers insights into the overall efficiency and security posture of applications developed under these languages.

The analysis shows that while Java has the highest number of code smells, it also has a significant foundation that may contribute to higher Minor and Major issues. Kotlin, with its efficient but critical-prone codebase, offers a middle ground. In contrast, Flutter maintains a large codebase with a lower incidence of critical issues, proving its robustness in handling complex applications. This comprehensive view helps underline the strengths and weaknesses of each language, guiding developers in choosing the most suitable technology based on the specific needs of their projects.

V Discussion

This chapter delves into the implications of the comparative analysis of mobile application development using Java, Kotlin, and Dart (Flutter), as presented in the previous results section. The findings underscore significant development practices, code quality, and maintainability disparities across these programming languages. The discussion further explores strategic enhancements that could be considered for future research and practical application development.

5.1 Interpretation of Results

The analysis revealed that while robust and widely used, Java exhibited the highest number of code smells and lines of code (LOC), which may complicate maintenance and scalability. Kotlin and Dart, however, demonstrated a more efficient coding practice by requiring fewer lines of code and presenting fewer critical issues. This efficiency reduces the potential for bugs and enhances the maintainability of applications. For instance, Kotlin's 48,865 LOC and Dart's 174,044 LOC contrast sharply with Java's 52,901 LOC, reflecting a more concise coding structure in newer languages, potentially leading to better manageability.

The SonarQube analysis of open-source projects further illuminated this point. Java projects displayed a substantial prevalence of minor and major issues, with a significant 65% being minor but still having a notable 9.5% critical issues. In contrast, Kotlin, despite a lower total LOC, showed a higher percentage of critical issues (18.9%), suggesting areas needing stringent quality checks. Dart (Flutter), however, managed a larger codebase with a relatively balanced distribution of code smells, maintaining fewer critical issues (6.1%), which underscores its capability to handle complex applications with better security and stability languages.

5.2 Recommendations for Future Development Practices

Given the findings, it is recommended that development teams consider adopting newer programming languages like Kotlin and Dart for their projects, especially when developing complex, scalable applications. These languages offer syntactical efficiencies and have advanced features that can significantly reduce the occurrence of code smells and other quality issues. Moreover, the study highlights the need for more efficient code quality assessment and application development methodologies. Incorporating machine learning models to predict potential code smells and automate code reviews could significantly enhance the efficiency and effectiveness of the development process. This approach not only saves time but also ensures higher standards of code quality.

5.3 Proposed Enhancements for Future Research

To extend the reliability and applicability of the findings from this study, future research should consider the following enhancements:

- **Inclusion of More Programming Languages:** Expanding the analysis to include other emerging languages and frameworks could provide a broader perspective on the optimal tools for various development needs.
- **Integration of Additional Metrics:** Besides LOC and code smells, incorporating metrics such as runtime efficiency, user experience, and energy consumption could provide a more comprehensive understanding of each language's performance.
- **Larger Sample Size:** Analyzing a broader array of projects from diverse repositories could help validate the current findings across different contexts and applications, potentially revealing new insights into the best practices in mobile application development.

5.4 Reflection on Methodological Approaches

The methodological approach of using SonarQube for individual projects and a cross-section of open-source projects has proven effective in highlighting differences in code quality across languages. However, analysis tools must be continuously updated and customized to keep pace with these languages' evolving features. For instance, the community-developed SonarQube plugin for Flutter was crucial in adapting the analysis tool to new technologies, underscoring the importance of community involvement in research methodologies.

VI Conclusion

This thesis has provided a comprehensive comparative analysis of mobile application development across three primary programming languages: Java, Kotlin, and Dart (Flutter). Through the detailed examination of each language's performance in developing a Kanban board application and the subsequent analysis of open-source projects using SonarQube, significant insights into the coding efficiencies, maintainability, and overall code quality have been gleaned. The findings underscore each language's inherent strengths and weaknesses, offering a nuanced understanding of their suitability for various development contexts.

Remarkably, the analysis of the Kanban board application development and the SonarQube assessments of open-source projects revealed that Kotlin and Dart exhibited superior maintainability and code quality compared to Java. Kotlin had a significantly lower lines of code (LOC) at 1,467 for the Kanban board and 48,865 for open-source projects, with fewer code smells (24 in the Kanban board and 647 in open-source projects) and a higher proportion of critical issues. Dart (Flutter) demonstrated an ability to manage larger codebases effectively, with 1,515 LOC for the Kanban board and 174,044 LOC for open-source projects, and a lower proportion of critical code smells (22 in the Kanban board and 1,303 in open-source projects). Java, despite its robustness, showed the highest number of code smells, indicating potential challenges in maintenance and scalability, with 1,748 LOC for the Kanban board and 52,901 LOC for open-source projects, and 62 code smells in the Kanban board and 3,590 code smells in open-source projects.

When selecting a programming language, it is crucial to consider project requirements, time scope, and developer experience. The language should be chosen based on the project's specific needs, such as the intended platform, complexity of the app, and feature requirements. The timeline for development and deployment must also be considered, and languages that allow for fast development and deployment might be preferable in time-sensitive projects. Finally, the development team's experience and familiarity with a particular language can significantly impact the efficiency and quality of the development process. Therefore, selection should be made considering the team's proficiency in a given language, making the transition easier.

The thesis findings suggest that developers should adopt newer programming languages like Kotlin and Dart to enhance their coding practices, as they have the potential to increase efficiency and reduce code complexity. Additionally, it is recommended that they utilize advanced tools for code analysis and quality assurance, such as SonarQube, to identify and address code smells and other quality issues regularly. Continuous learning and training are also essential for developers to keep up-to-date with the latest developments in programming languages and frameworks, which can help improve the maintainability of their applications.

Future research should explore the integration of additional programming languages and newer frameworks to update our understanding of best practices in application development continually. Moreover, adopting more comprehensive metrics, including runtime efficiency and user experience, could provide a broader perspective on the performance of different languages. Extending the analysis to include a larger dataset of open-source projects could also enhance the generalizability of the findings.

This thesis highlights the importance of carefully selecting a mobile application development language based on specific project requirements, the time scope for development, and the development team's experience. While newer languages like Kotlin and frameworks like Flutter offer substantial benefits in terms of efficiency and maintainability, the choice of technology should always be tailored to the project's specific needs and the development team's capabilities. Through thoughtful selection and implementation of development languages, developers can optimize their processes, enhance the quality of their applications, and ultimately, contribute to the advancement of mobile technology. This research advocates for a balanced approach to framework selection, emphasizing continuous quality assurance and adopting best practices as pivotal to the long-term success of mobile application development projects.

Reference

[1] Statista. Number of Mobile App Downloads Worldwide from 2019 to 2027, by Segment (in million downloads) [Graph]. https://www.statista.com/forecasts/1262881/mobile-app-download-worldwide-by-segment, 2023. Accessed: 22-May-2024.

[2] JetBrains. Cross-Platform Mobile Frameworks Used by Software Developers Worldwide from 2019 to 2022 [Graph]. https://www.statista.com/statistics/869224/worldwide-software-developer-working-hours/, 2022. [Online; accessed 22-May-2024].

[3] Yuhua Li, Xiheng Gong, Jingyi Zhang, Ziwei Xiang, and Chengjun Liao. The Impact of Mobile Payment on Household Poverty Vulnerability: A Study Based on CHFS2017 in China. *International Journal of Environmental Research and Public Health*, 19(21):14001, 2022.

[4] Quistina Omar, Ching Seng Yap, Poh Ling Ho, and William Keling. Predictors of Behavioral Intention to Adopt E-AgriFinance App Among the Farmers in Sarawak, Malaysia. *British Food Journal*, 124(1):239–254, 2022.

[5] Xue Wang. Mobile Payment and Informal Business: Evidence from China's Household Panel Data. *China & World Economy*, 28(3):90–115, 2020.

[6] Egidijus Rybakovas and Gerda Zigiene. Financial Innovation for Financial Inclusion:

Mapping Potential Access to Finance. In *European Conference on Innovation and Entrepreneurship*, volume 17, pages 451–457, 2022.

[7] Jennifer Dahne and Carl W Lejuez. Smartphone and Mobile Application Utilization Prior to and Following Treatment among Individuals Enrolled in Residential Substance use Treatment. *Journal of substance abuse treatment*, 58:95–99, 2015.

[8] Yingxin Zhang, Yijing Du, and Yan Li. Entertainment Apps, Limited Attention and Investment Performance. *Frontiers in Psychology*, 14:1118797, 2023.

[9] Andreas Biørn-Hansen, Christoph Rieger, Tor-Morten Grønli, Tim A Majchrzak, and Gheorghita Ghinea. An Empirical Investigation of Performance Overhead in Cross-Platform Mobile Development Frameworks. *Empirical Software Engineering*, 25:2997–3040, 2020.

[10] Kamil Wasilewski and Wojciech Zabierowski. A Comparison of Java, Flutter and Kotlin/Native Technologies for Sensor Data-Driven Applications. *Sensors*, 21(10):3324, 2021.

[11] Luca Ardito, Riccardo Coppola, Giovanni Malnati, and Marco Torchiano. Effectiveness of Kotlin vs. Java in Android App Development Tasks. *Information and Software Technology*, 127:106374, 2020.

[12] Riccardo Coppola, Luca Ardito, and Marco Torchiano. Characterizing the Transition to Kotlin of Android Apps: a Study on F-droid, Play Store, and Github. In *Proceedings of the 3rd ACM SIGSOFT International Workshop on App Market Analytics*, pages 8–14, 2019.

[13] Riofebri Prasetia Prasetia and Lutfi Rahmatuti Maghfiroh Maghfiroh. Development of FASIH Application for the Badan Pusat Statistisk using Flutter Framework. In *Proceedings of The International Conference on Data Science and Official Statistics*, pages 798–809, 2023.

[14] Dieter Meiller. Flutter: The Future of Application Development? https://www.iadisportal.org/digital-library/flutter-the-future-of-application-development, 2022. Accessed: 22-May-2024.

[15] Rajiv D Banker, Gordon B Davis, and Sandra A Slaughter. Software development practices, software complexity, and software maintenance performance: A field study. *Management science*, 44(4):433–450, 1998.

[16] Daniel Graziotin, Xiaofeng Wang, and Pekka Abrahamsson. Happy Software Developers Solve Problems Better: Psychological Measurements in Empirical Software Engineering. *PeerJ*, 2:e289, 2014.

[17] Manish Agrawal and Kaushal Chari. Software Effort, Quality, and Cycle Time: A Study of CMM Level 5 Projects. *IEEE Transactions on software engineering*, 33(3):145–156, 2007.

[18] Daniel Graziotin, Fabian Fagerholm, Xiaofeng Wang, and Pekka Abrahamsson. What Happens When Software Developers are (un) happy. *Journal of Systems and Software*, 140:32–47, 2018.

[19] Thos Stamford Raffles. The History of Java. *The Monthly Magazine*, 43(300):598–622, 1817.

[20] Gustavo Pinto, Weslley Torres, Benito Fernandes, Fernando Castor, and Roberto SM Barros. A Large-Scale Study on the Usage of Java's Concurrent Programming Constructs. *Journal of Systems and Software*, 106:59–81, 2015.

[21] Li Li, Tegawendé F Bissyandé, Yves Le Traon, and Jacques Klein. Accessing Inaccessible Android Apis: An Empirical Study. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 411–422. IEEE, 2016.

[22] Paul King. A History of the Groovy Programming Language. *Proceedings of the ACM on Programming Languages*, 4(HOPL):1–53, 2020.

[23] Lu Li and Yan Liu. Mapping Modern JVM Language Code to Analysis-friendly Graphs: A Pilot Study with Kotlin. In *SEKE*, pages 67–72, 2022.

[24] Dwi Kartinah. Android-Based Health Post Management Application Design Clinics in Indonesia. *International Journal Science and Technology*, 2(2):36–41, 2023.

[25] Nardjes Bouchemal, Aissa Serrar, Yehya Bouzeraa, and Naila Bouchmemal. Scream to Survive (S2S): Intelligent System to Life-Saving in Disasters Relief. In *Machine Learning for Networking: Second IFIP TC 6 International Conference, MLN 2019, Paris, France, December 3–5, 2019, Revised Selected Papers 2*, pages 414–430. Springer, 2020.

[26] Siti Ernawati and Risa Wati. Android-Based Quran Application On The Flutter Framework By Using The Fountain Model. *Jurnal Riset Informatika*, 3(2):195–202, 2021.

[27] Ardhya Pandu Pratama and Made Kamisutara. Pengembangan Sistem Informasi Akademik Berbasis Mobile Menggunakan Flutter Di Universitas Narotama Surabaya. *Jurnal Ilmiah NERO*, 6(2):145–160, 2021.

[28] Péter Hegedűs and Rudolf Ferenc. Static Code Analysis Alarms Filtering Reloaded: A New Real-World Dataset and its ML-Based Utilization. *IEEE Access*, 10:55090–55101, 2022.

[29] Alejandro Mazuera-Rozo, Camilo Escobar-Velásquez, Juan Espitia-Acero, David Vega-Guzmán, Catia Trubiani, Mario Linares-Vásquez, and Gabriele Bavota. Taxonomy of Security Weaknesses in Java and Kotlin Android Apps. *Journal of systems and software*, 187:111233, 2022.

[30] Victor Oliveira, Leopoldo Teixeira, and Felipe Ebert. On the Adoption of Kotlin on Android Development: A Triangulation Study. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 206–216. IEEE, 2020.

[31] Matheus Flauzino, Júlio Veríssimo, Ricardo Terra, Elder Cirilo, Vinicius HS Durelli, and Rafael S Durelli. Are You Still Smelling it? A comparative Study Between Java and Kotlin Language. In *Proceedings of the VII Brazilian symposium on software components, architectures, and reuse*, pages 23–32, 2018.

[32] Geoffrey Hecht, Romain Rouvoy, Naouel Moha, and Laurence Duchien. Detecting Antipatterns in Android Apps. In *2015 2nd ACM international conference on mobile software engineering and systems*, pages 148–149. IEEE, 2015.

[33] Maxime Lamothe, Weiyi Shang, and Tse-Hsun Peter Chen. A3: Assisting Android Api Migrations Using Code Examples. *IEEE Transactions on Software Engineering*, 48(2):417–431, 2020.

[34] MAA Mamun, Miroslaw Staron, Christian Berger, Regina Hebig, and Jörgen Hansson. Improving Code Smell Predictions in Continuous Integration by Differentiating Organic from Cumulative Measures. In *The Fifth International Conference on Advances and Trends in Software Engineering*, pages 62–71.

[35] Diego Marcilio, Rodrigo Bonifácio, Eduardo Monteiro, Edna Canedo, Welder Luz, and Gustavo Pinto. Are Static Analysis Violations Really Fixed? a Closer Look at Realistic Usage of Sonarqube. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pages 209–219. IEEE, 2019.

[36] Ifeanyi Rowland Onyenweaku, Michael Scott Brown, Michael Pelosi, and MH Shahine. A Sonarqube Static Analysis of the Spectral Workbench. *International Journal of Natural Science and Reviews*, 6(16):1–15, 2021.

[37] Sebastian Stiernborg. Automated Code Inspection: Investigating Deployment of Continuous Inspection, 2019.

[38] Zéphyrin Soh, Aiko Yamashita, Foutse Khomh, and Yann-Gaël Guéhéneuc. Do Code Smells Impact the Effort of Different Maintenance Programming Activities?

In *2016 IEEE 23Rd international conference on software analysis, evolution, and reengineering (SANER)*, volume 1, pages 393–402. IEEE, 2016.

[39] Luis Corral and Ilenia Fronza. Better Code for Better Apps: A Study on Source Code Quality and Market Success of Android Applications. In *2015 2nd ACM International Conference on Mobile Software Engineering and Systems*, pages 22–32. IEEE, 2015.

[40] Tarek Alkhaeir and Bartosz Walter. The Effect of Code Smells on the Relationship Between Design Patterns and Defects. *IEEE Access*, 9:3360–3373, 2020.

[41] Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, and Denys Poshyvanyk. When and Why Your Code Starts to Smell Bad. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 403–414. IEEE, 2015.

[42] Xiaofeng Han, Amjed Tahir, Peng Liang, Steve Counsell, and Yajing Luo. Understanding Code Smell Detection via Code Review: A Study of the Openstack Community. In *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*, pages 323–334. IEEE, 2021.

[43] Yoonsik Cheon and Carlos Chavez. Converting Android Native Apps to Flutter Cross-Platform Apps. In *2021 International conference on computational science and computational intelligence (CSCI)*, pages 1898–1904. IEEE, 2021.

[44] Piotr Nawrocki, Krzysztof Wrona, Mateusz Marczak, and Bartlomiej Sniezynski. A Comparison of Native and Cross-Platform Frameworks for Mobile Applications. *Computer*, 54(3):18–27, 2021.

[45] Bloc Library. Bloc: A Predictable State Management Library for Dart. https://bloclibrary.dev, 2024. Accessed: 22-May-2024.

[46] Thorn Chorn. GetX Documentation. https://chornthorn.github.io/getx-docs/, 2024. Accessed: 22-May-2024.

[47] Remi Rousselet. Provider Package. https://pub.dev/packages/provider, 2024. Accessed: 22-May-2024.

[48] Robert C Martin. *Clean Code: a Handbook of Agile Software Craftsmanship*. Pearson Education, 2009.

[49] Jaykishan Sewak. MVVM Architecture in Android Using Kotlin: A Practical Guide. https://medium.com/@jecky999/mvvm-architecture-in-android-using-kotlin-a-practical-guide-73f8de1d9c58, Jun 2023. Accessed: 22-May-2024.

[50] Valentina Lenarduzzi, Nyyti Saarimäki, and Davide Taibi. The Technical Debt Dataset. In *Proceedings of the fifteenth international conference on predictive models and data analytics in software engineering*, pages 2–11, 2019.

[51] Jie Tan, Daniel Feitosa, Paris Avgeriou, and Mircea Lungu. Evolution of Technical Debt Remediation in Python: A Case Study on the Apache Software Ecosystem. *Journal of Software: Evolution and Process*, 33(4):e2319, 2021.

[52] Simon Kawuma and Evarist Nabaasa. An Empirical Study of Bugs in Eclipse Stable Internal Interfaces. *arXiv preprint arXiv:2203.09134*, 2022.

[53] Rubén Saborido, Javier Ferrer, Francisco Chicano, and Enrique Alba. Automatizing Software Cognitive Complexity Reduction. *IEEE Access*, 10:11642–11656, 2022.

[54] Irene Tollin, Francesca Arcelli Fontana, Marco Zanoni, and Riccardo Roveda. Change Prediction Through Coding Rules Violations. In *Proceedings of the 21st international conference on evaluation and assessment in software engineering*, pages 61–64, 2017.

[55] Daniel Guaman and Paola Cabrera Barba-Guamán. SonarQube as a Tool to Identify Software Metrics and Technical Debt.

[56] Docker, Inc. Docker Documentation. https://docs.docker.com/, 2024. Accessed: 22-May-2024.

[57] Insideapp-Oss. SonarQube plugin for Flutter / Dart. https://github.com/insideapp-oss/sonar-flutter, 2024. Accessed: 22-May-2024.

[58] SonarQube Web API Documentation. https://docs.sonarsource.com/sonarqube/latest/extension-guide/web-api/, 2024. Accessed: 22-May-2024.

Appendices

I Project Listings for Automated SonarQube Analysis

Table 1.1: Java Open Source Projects Selected for Analysis

Project Name	GitHub Link
KanbanBoardApp	https://github.com/Sardor6628/Kanban-Board-Java
CountdownTimer2	https://github.com/sagarsiddhpura/CountdownTimer
Easy-Attendance-App	https://github.com/jaikeerthick/Easy-Attendance-App
StopWatchRemade	https://github.com/0developers/StopWatchRemade
Post-it-Notes-App	https://github.com/harshkv/Post-it-Notes-App
VideoPlayer	https://github.com/waleedtalha/VideoPlayer
CurrencyConverter	https://github.com/Ch-Tima/CurrencyConverter/tree/master
passwordgenerator	https://github.com/SecUSo/privacy-friendly-passwordgenerator
finance-manager	https://github.com/SecUSo/privacy-friendly-finance-manager
todo-list	https://github.com/SecUSo/privacy-friendly-todo-list
pain-diary	https://github.com/SecUSo/privacy-friendly-pain-diary
qr-scanner	https://github.com/SecUSo/privacy-friendly-qr-scanner
tape-measure	https://github.com/SecUSo/privacy-friendly-tape-measure
pedometer	https://github.com/SecUSo/privacy-friendly-pedometer
weather	https://github.com/SecUSo/privacy-friendly-weather

Table 1.2: Kotlin Open Source Projects Selected for Analysis

Project Name	GitHub Link
KanbanBoa	https://github.com/Sardor6628/Kanban-Board-Kotlin
vocable-android	https://github.com/willowtreeapps/vocable-android?tab=readme-ov-file
plees-tracker	https://github.com/vmiklos/plees-tracker
NotyKT	https://github.com/PatilShreyas/NotyKT
muzei	https://github.com/muzei/muzei
pdf-viewer-pro	https://github.com/Sav22999/sav-pdf-viewer-pro
vocable	https://github.com/PhilT95/ger-es_trainer
MyNotes	https://github.com/akshatbhuhagal/MyNotes
RecurringExpenseTracker	https://github.com/DennisBauer/RecurringExpenseTracker?tab=readme-ov-file
Simple-Dialer	https://github.com/SimpleMobileTools/Simple-Dialer
PhoneBook	https://github.com/KishanViramgama/PhoneBook_CRUD
SoundMeter1	https://github.com/albertopasqualetto/SoundMeterESP
Simple-Draw	https://github.com/SimpleMobileTools/Simple-Draw
unitconverter	https://github.com/dbrant/unitconverter-android/tree/master
Simple-Voice-Recorder	https://github.com/SimpleMobileTools/Simple-Voice-Recorder/tree/master

Table 1.3: Dart Open Source Projects Selected for Analysis

Project Name	GitHub Link
KanbanBoardApp	https://github.com/Sardor6628/kanban_board_crm
day-night-time-picker	https://github.com/subhamayd2/day_night_time_picker.git
android-tv-app	https://github.com/mawaqit/android-tv-app.git
ConsumerFlutterApp	https://github.com/LaCoro/ConsumerFlutterApp.git
flutter-chat-craft	https://github.com/taxze6/flutter-chat-craft.git
Flutter-TDD-Clean-Architecture-E-Commerce-App	https://github.com/Sameera-Perera/Flutter-TDD-Clean-Architecture-E-Commerce-App.git
flutter-samples	https://github.com/yuto-yuto/flutter_samples.git
flutter-unit	https://github.com/toly1994328/FlutterUnit
flutter-quill	https://github.com/singerdmx/flutter-quill.git
Musify	https://github.com/gokadzev/Musify.git
Personal-Finance-Manager	https://github.com/sajitha00/Personal-Finance-Manager.git
mobile	https://github.com/realm/realm-dart
spotube	https://github.com/KRTirtho/spotube.git
anonaddy	https://github.com/KhalidWar/anonaddy
pstube	https://github.com/prateekmedia/pstube.git

II Python Script for Project Automation and Configuration

Algorithm 2.1: Python Scripts for Project Automation and Configuration

```
import subprocess
import os
import requests
import base64
def encode_credentials(username, password):
    credentials = f"{username}:{password}"
    encoded_credentials = base64.b64encode(credentials.encode()).
         decode()
    return encoded_credentials


def create_sonar_project(project_key, project_name,
 encoded_credentials):
    url = f'http://localhost:9000/api/projects/create?project={
         project_key}&name={project_name}'
    headers = {
        'Authorization': f'Basic {encoded_credentials}',
        'Content-Type': 'application/x-www-form-urlencoded'
    }
    response = requests.post(url, headers=headers)
    return response


def generate_sonar_token(project_key, token_name, encoded_credentials
 ):
    url = f'http://localhost:9000/api/user_tokens/generate?name={
         token_name}&projectKey={project_key}&type=
         PROJECT_ANALYSIS_TOKEN'
    headers = {
        'Authorization': f'Basic {encoded_credentials}',
```

```python
            'Content-Type': 'application/x-www-form-urlencoded'
        }
        response = requests.post(url, headers=headers)
        return response


def clone_and_setup_sonar(repo_urls, save_directory, username, password):
    encoded_credentials = encode_credentials(username, password)
    os.makedirs(save_directory, exist_ok=True)

    for url in repo_urls:
        repo_name = url.split('/')[-1].replace('.git', '')
        clone_path = os.path.join(save_directory, repo_name)
        config_file_path = os.path.join(clone_path, 'sonarqube_configuration.txt')
        config_file_properties = os.path.join(clone_path, 'sonar-project.properties')


        # Skip cloning if repo is already cloned and configuration file exists
        if os.path.exists(clone_path) and os.path.exists(config_file_path):
            print(f'Repository "{repo_name}" already cloned and configured. Skipping...')
            continue

        # Clone repo if it doesn't exist
        if not os.path.exists(clone_path):
            result = subprocess.run(['git', 'clone', url, clone_path], stdout=subprocess.PIPE, stderr=subprocess.PIPE,
                                    universal_newlines=True)
            if result.returncode != 0:
                print(f'Failed to clone {repo_name}. Error: {result.stderr}')
                continue

        # Proceed with SonarQube project creation and token generation
        print(f'Cloned {repo_name}')
        project_response = create_sonar_project(repo_name, repo_name, encoded_credentials)
        if project_response.status_code == 200:
            print(f'SonarQube project created for {repo_name}')
            token_response = generate_sonar_token(repo_name, f"{repo_name}_token", encoded_credentials)
            if token_response.status_code == 200 and 'token' in token_response.json():
                token = token_response.json()['token']
                with open(config_file_properties, 'w') as properties_file:
                    properties_file.write(f"""
                    #Project identification
                    sonar.projectKey={repo_name}
                    sonar.projectName={repo_name}
                    sonar.projectVersion=1.0

                    # Source code location.
                    # Path is relative to the sonar-project.properties file. Defaults to .
                    # Use commas to specify more than one file/folder.
                    # It is good practice to add pubspec.yaml to the sources as the analyzer
                    # may produce warnings for this file as well.
                    sonar.sources=lib,pubspec.yaml
                    sonar.tests=test
```

```

                        # Encoding of the source code. Default is default
                              system encoding.
                        sonar.sourceEncoding=UTF-8
                        """)
                    with open(config_file_path, 'w') as token_file:
                        token_file.write(f"""\n#
                        #Project identification
                        sonar.projectKey={repo_name}
                        sonar.projectName={repo_name}
                        sonar.projectVersion=1.0

                        # Source code location.
                        # Path is relative to the sonar-project.
                             properties file. Defaults to .
                        # Use commas to specify more than one file/folder
                             .
                        # It is good practice to add pubspec.yaml to the
                             sources as the analyzer
                        # may produce warnings for this file as well.
                        sonar.sources=lib,pubspec.yaml
                        #sonar.tests=test

                        # Encoding of the source code. Default is default
                              system encoding.
                        sonar.sourceEncoding=UTF-8


                        run following:

                    sonar-scanner \
      -Dsonar.projectKey={repo_name} \
      -Dsonar.sources=. \
      -Dsonar.host.url=http://localhost:9000 \
      -Dsonar.login={token}

                                                    """)

                    print(f'SonarQube configuration saved to {
                         config_file_path}')
                else:
                    print(f'Failed to generate SonarQube token for {
                         repo_name}.')
            else:
                print(f'Failed to create SonarQube project for {repo_name
                     }.')

        print('Finished processing repositories and setting up SonarQube
             projects.')


    if __name__ == '__main__':
        repo_urls = [
            #Flutter projects
            "https://github.com/shiosyakeyakini-info/miria.git",
            "https://github.com/hoc081098/hoc081098.git",
            "https://github.com/singerdmx/flutter-quill.git"
            "https://github.com/deckerst/aves.git",
            "https://github.com/clragon/e1547.git",
            "https://github.com/realm/realm-dart",
            "https://github.com/KhalidWar/anonaddy",
            "https://github.com/prateekmedia/pstube.git"
        ]
        save_directory = 'flutter_repositories'
        username = 'admin'  # SonarQube username
        password = 'superadmin'  # SonarQube password
        clone_and_setup_sonar(repo_urls, save_directory, username,
             password)
```

III Python Script for Data Retrieval and Report Generation

Algorithm 3.1: Python Script for Data Retrieval and Report Generation label

```python
import requests
import pandas as pd

pd.set_option('display.max_rows', None)
pd.set_option('display.max_columns', None)
pd.set_option('display.width', 1000)
pd.set_option('display.max_colwidth', None)

def fetch_project_metrics_and_issues_summary(base_url, projects, types="
    CODE_SMELL", ps=500):
    project_summaries = []

    for project in projects:
        # Fetch issues summary
        issues_params = {
            "componentKeys": project,
            "types": types,
            "ps": ps
        }

        issues_response = requests.get(f"{base_url}/api/issues/search",
             params=issues_params)

        # Fetch project metrics
        metrics_params = {
            "component": project,
            "metricKeys": "ncloc,complexity,violations"
        }

        metrics_response = requests.get(f"{base_url}/api/measures/
             component", params=metrics_params)

        if issues_response.status_code == 200 and metrics_response.
             status_code == 200:
            issues_data = issues_response.json()
            metrics_data = metrics_response.json()

            # Parse issues data
            issues = issues_data.get("issues", [])
            severity_counts = {
                "BLOCKER": 0,
                "CRITICAL": 0,
                "MAJOR": 0,
                "MINOR": 0,
                "INFO": 0
            }
            for issue in issues:
                if issue["severity"] in severity_counts:
                    severity_counts[issue["severity"]] += 1

            # Parse metrics data for ncloc (Lines of Code)
            measures = metrics_data["component"]["measures"]
            loc = next((measure["value"] for measure in measures if
                 measure["metric"] == "ncloc"), "0")

            # Prepare the summary for the current project
            project_summary = {
                "NAME": project,
                "BLOCKER": severity_counts["BLOCKER"],
                "CRITICAL": severity_counts["CRITICAL"],
                "MAJOR": severity_counts["MAJOR"],
                "MINOR": severity_counts["MINOR"],
                "INFO": severity_counts["INFO"],
```

```python
                "total": len(issues),
                "LOC": loc  # Add LOC to the project summary
            }

            project_summaries.append(project_summary)
        else:
            print(
                f"Failed to fetch data for project {project}, issues status code: {issues_response.status_code}, metrics status code: {metrics_response.status_code}")

    return pd.DataFrame(project_summaries)


base_url = "http://localhost:9000"
projects = ["Kanban_board_flutter","day_night_time_picker", "android-tv-app", "ConsumerFlutterApp", "flutter-chat-craft",
            "Flutter-TDD-Clean-Architecture-E-Commerce-App", "flutter_samples", "Kanban_board_flutter", "flutter-quill",
            "Musify", "openfoodfacts-dart", "Personal-Finance-Manager",
            "mobile", "spotube", "anonaddy", "pstube"]
project_metrics_and_issues_summary_df = fetch_project_metrics_and_issues_summary(base_url, projects)
# Display the table
print(project_metrics_and_issues_summary_df)

# Save the DataFrame to a CSV file
report_path = "flutter-report.csv"
project_metrics_and_issues_summary_df.to_csv(report_path, index=False)
print(f"Flutter Report saved to {report_path}")

projects = ["Kanban_board_java", 'CountdownTimer2', '2048-android2', 'Easy-Attendance-App', 'StopWatchRemade',
            'Post-it-Notes-App', 'VideoPlayer', 'CurrencyConverter', 'passwordgenerator', 'finance-manager',
            'todo-list', 'pain-diary', 'qr-scanner', 'tape-measure', 'pedometer', 'weather']

project_metrics_and_issues_summary_df = fetch_project_metrics_and_issues_summary(base_url, projects)
# Display the table
print(project_metrics_and_issues_summary_df)

# Save the DataFrame to a CSV file
report_path = "java-report.csv"
project_metrics_and_issues_summary_df.to_csv(report_path, index=False)
print(f"Java Report saved to {report_path}")

projects = ["Kanban_board_kotlin", "vocable-android", "plees-tracker", "NotyKT",
            "muzei", "awaker", "GitExplorer-Android",
            'pdf-viewer-pro', 'vocable', 'MyNotes',
            'RecurringExpenseTracker',
            'Simple-Dialer', 'PhoneBook',
            'SoundMeter1', 'Simple-Draw', 'unitconverter', 'Simple-Voice-Recorder']

project_metrics_and_issues_summary_df = fetch_project_metrics_and_issues_summary(base_url, projects)
# Display the table
print(project_metrics_and_issues_summary_df)

# Save the DataFrame to a CSV file
report_path = "kotlin-report.csv"
project_metrics_and_issues_summary_df.to_csv(report_path, index=False)
print(f"Kotlin Report saved to {report_path}")
```

IV Implementation Details of Task Models

Algorithm 4.1: Java Task Model Implementation

```java
package com.example.kanban_board_java.data.response;
import com.google.firebase.firestore.Exclude;
public class TaskResponse {
    private String id;
    @Exclude
    private String documentId;
    private String title;
    private String userId;
    private String description;
    private Long createdTime;
    private Long completedTime;
    private Long startedTime;
    private long spentTime;
    private String currentStatus;
    public String getId() {
        return id;}
    public void setId(String id) {
        this.id = id;}
    public String getDocumentId() {
        return documentId;}
    public void setDocumentId(String documentId) {
        this.documentId = documentId;   }
    public String getTitle() {
        return title;}
    public void setTitle(String title) {
        this.title = title;}
    public String getUserId() {
        return userId;}
    public void setUserId(String userId) {
        this.userId = userId;}
    public String getDescription() {
        return description;}
    public void setDescription(String description) {
        this.description = description;     }
    public Long getCreatedTime() {
        if (createdTime == null) {
            return 1000L;    }
        return createdTime;}
    public void setCreatedTime(Long createdTime) {
        this.createdTime = createdTime; }
    public Long getCompletedTime() {
        if (completedTime == null) {
            return 1000L;}
        return completedTime;}
    public void setCompletedTime(Long completedTime) {
        this.completedTime = completedTime; }
    public Long getStartedTime() {
        if (startedTime == null) {
            return 1000L;     }
        return startedTime;}
    public void setStartedTime(Long startedTime) {
        this.startedTime = startedTime;      }
    public long getSpentTime() {
        return spentTime;}
    public void setSpentTime(long spentTime) {
        this.spentTime = spentTime;}
    public String getCurrentStatus() {
        return currentStatus;}
    public void setCurrentStatus(String currentStatus) {
        this.currentStatus = currentStatus;  }}
```

Algorithm 4.2: Kotlin Task Model Implementation

```kotlin
package com.example.kanban_board_java_kotlin.data.response
import android.os.Parcelable
import com.google.firebase.firestore.Exclude
import kotlinx.parcelize.Parcelize
@Parcelize
data class TaskResponse(
    var id: String = System.currentTimeMillis().toString(),
    @get:Exclude var documentId: String = "",
    var title: String = "",
    var userId: String = "",
    var description: String = "",
    var createdTime: Long? = null,
    var completedTime: Long? = null,
    var startedTime: Long? = null,
    var spentTime: Long = 0,
    var currentStatus: String = "todo"
) : Parcelable
```

Algorithm 4.3: Dart Task Model Implementation

```
import 'package:kanban_board/constants/contant_variables.dart';
class Task {
  String id;
  String title;
  String userId;
  String description;
  DateTime? createdTime;
  DateTime? completedTime;
  DateTime? startedTime;
  int spentTime;
  String currentStatus;
  Task({
    required this.id,
    required this.title,
    required this.userId,
    required this.description,
    required this.createdTime,
    required this.completedTime,
    required this.startedTime,
    required this.spentTime,
    required this.currentStatus,});}
```

교차 플랫폼 및 네이티브 모바일 앱 개발 접근 방식의 비교 분석

이브로키모브 사도르벡

부산대학교 대학원 정보융합공학과

요약

모바일 앱 개발 방법에 대해 많은 접근법이 제안되었지만, 개발자들은 적합한 방법 선택에 어려움을 겪고 있다. 이 연구는 네이티브 및 크로스 플랫폼 어플리케이션 개발 접근 방식을 비교하며, 특히 Java에서 Kotlin으로의 선호도 변화와 Flutter의 사용 증가에 중점을 두고 비교한다. 이 연구는 Java, Kotlin, Dart(Flutter)를 사용하여 같은 어플리케이션을 작성함으로써, 모바일 어플리케이션 개발에서 프로그래밍 언어 및 프레임워크 선택에 영향을 미치는 요인에 관한 실질적인 통찰을 제공한다. 또한, 이 연구는 45개의 GitHub 공개 소스 프로그램의 코드 품질도 검사함으로써 모범 개발 사례를 모색한다. 구체적으로, 반자동 SonarQube 평가를 사용하여 LOC 및 코드 스멜을 평가하고, 이를 통해 특정 언어나 프레임워크 선택에 따라 코드의 유지보수 및 개발 효율성에 미치는 영향을 탐구한다. 실험 결과, 작성된 코드 품질의 차이를 알 수 있었다. 본 연구는 코드 스멜을 줄이고 프로젝트 유지보수를 개선하는 데 적합한 프로그래밍 언어와 프레임워크를 선택하는 데 도움이 될 것으로 기대된다.