교차 플랫폼 및 네이티브 모바일 앱 개발 접근 방식의 비교 분석

이브로키모브 사도르벡 [1], 우균 [2]
[1] 부산대학교 정보융합공학과
[2] 부산대학교 정보컴퓨터공학부

{sardor, woogyun}@pusan.ac.kr

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

Ibrokhimov Sardorbek Rustam Ugli[1], Gyun Woo[2]
[1]Dept. of Information Convergence Engineering, Pusan National University
[2]School of Computer Science and Engineering, Pusan National University

Abstract

Though lots of approaches to develop mobile apps are suggested up to now, developers have difficulties selecting a 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 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 how to best optimize language and framework selection to reduce code smells and improve project maintainability.

1. Introduction

In the field of mobile application development, the choice of programming language and development framework significantly impacts the efficiency, maintainability, and performance of applications. While Java has traditionally been dominant in Android development, there has been a notable shift towards Kotlin due to its concise syntax, safety features, and interoperability with Java [1]. This transition to Kotlin is significant as it enhances the development process and contributes to the overall quality of Android applications. The emergence of cross-platform frameworks such as Flutter, which allows for a single codebase across both Android and iOS, further revolutionizes development practices, although the adoption of such technologies must account for their impacts on code quality and maintainability, emphasizing the need for empirical studies to assess these effects [1].

How do Java, Kotlin, and Dart compare in development experience and code quality for the same application, particularly regarding code maintainability and development efficiency? This includes code smells and lines of code (LOC). The aim is to uncover differences and establish best practices for future strategies. The industry trend towards frameworks like Flutter, which streamline development and minimize separate codebases, responds to the needs of various users and promotes broader societal benefits through innovative approaches. However, the choice of language or framework often hinges on more than developer preference or speed, with vital factors like code quality and maintainability needing more empirical study [2][3].

This study aims to evaluate the development experience and code quality of creating a Kanban board application using Kotlin, Dart (Flutter), and Java, alongside assessing the prevalence of code smells in these frameworks across a broad sample of open-source projects. By analyzing the practical benefits and drawbacks of each development approach, the research seeks to understand the shift within the developer community from Java to Kotlin and the growing interest in Flutter for cross-platform development. Structured into six sections, the paper will discuss related research, describe the development and code quality evaluation processes using SonarQube, present the findings from data collection on code severity metrics, include a discussion section on the integration of community-developed tools and potential biases in code evaluation, and conclude with the implications of these results for future development practices.

2. Literature Review

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 [6]. 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 [7]. However, concerns persist regarding Flutter's performance compared to native applications and the maturity of its ecosystem.

The integration of tools like SonarQube in mobile app development underscores the importance of addressing technical debt and code smells [8]. Innovations like ecoCode represent the growing focus on energy-efficient coding, aligning with the broader trend of sustainable software development. Comparative studies on development methodologies reveal the challenges and benefits of transitioning between frameworks, providing valuable insights for developers [9]. Evaluating open-source projects yields insights into different development methodologies, helping to identify patterns, best practices, and common pitfalls across various languages and frameworks. However, there remains a research gap in combining practical development experiences with in-depth analysis of open-source projects to explore how methodologies influence code quality metrics such as code smells and their severity [10].

3. Methodology

3.1 Analysis Steps

In the methodology section of the paper, the analysis was conducted in three key steps:

Step 1. Project Implementation Across Frameworks: Development of an identical application using Kotlin, Flutter, and Java, focusing on comparing efficiencies and experiences across different app development strategies.

Step 2. Code Inspection: Utilization of SonarQube for code quality analysis, including the setup with Docker and a community-developed Dart plugin to facilitate comprehensive framework analysis.

Step 3. Severity Analysis and Assessment: Evaluation of issues identified during code inspection, categorized into severity levels—Blocker, Critical, Major, Minor, and Info—to help prioritize fixes based on their potential impact.

3.2 Project Implementation Across Frameworks

This section discusses the construction of a Kanban board app using Kotlin, Dart, and Java, aiming to compare the development experiences and efficiencies of cross-platform and native app development. The app features Kanban Board Visualization, Task Management, and Firebase Integration [12], highlighting implementation challenges such as third-party service integration and real-time data synchronization.

The project started with a simple UI designed in Figma, initially developed with Flutter in Android Studio using BLOC for state management [14]. This phase lasted 15-20 hours, facilitated by Firebase's compatibility with Google technologies.

Kotlin followed, applying the MVVM pattern [15] for a modular design, taking 20-24 hours due to less familiarity. Java, using Kotlin's XML and Android Studio's tools, required 3-4 hours but needed significantly more code: 21 LOC in Dart, 17 in Kotlin, and 84 in Java [16].

App size comparisons showed Java and Kotlin apps at 11.5MB, while Flutter's app was larger at 21.2MB. Performance and UI tests across mobile phones revealed minimal differences. This exploration serves as a practical comparison of development methodologies, platform-specific challenges, and strategic decision-making in mobile app development.

3.3 Code Inspection

SonarQube, an open-source tool by SonarSource, assesses code quality in 29 languages, identifying bugs and code smells [17]. In our Kanban project, we used SonarQube to categorize code smells by severity—Blocker, Critical, Major, Minor, and Info. Blocker issues are the most severe, potentially causing critical failures and require immediate attention. Critical issues, less severe, still need quick resolution, while Major and Minor issues impact code quality to a lesser extent but should be addressed. Info issues provide insights but don't demand urgent action. This severity ranking aids in prioritizing fixes. Metrics tracked included total code smells and lines of code (LOC). For Flutter compatibility, we installed Docker [18] and ran SonarQube in a Docker container with a Dart plugin for analysis [19].

3.4 Severity Analysis and Assessment.

<Table 1> SonarQube Report for Kanban Board app

	Java	Kotlin	Flutter
Blocker	0	0	0
Critical	16	7	0
Major	11	3	4
Minor	35	14	18
Info	0	0	0
Total	62	24	22

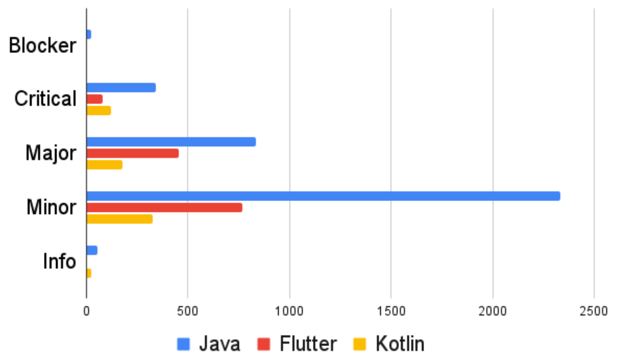
Customizing the analysis process for Flutter projects involved some adjustments. Initially, we manually registered each project in SonarQube, generating a unique token.

Subsequently, we set up our Java, Kotlin, and Flutter projects to compile reports for SonarQube submission. Kotlin and Dart showed similar LOC—1,467 and 1,515—contrasting with Java's higher 1,748 LOC. As detailed in Table 1, Both Kotlin and Dart had comparable Code Smell counts, 24 and 22, with Dart notably free of Critical issues, mainly presenting Minor and Major concerns affecting the project minimally. Java stood out with the most Code Smells—16 Critical, 22 Major, and 35 Minor—indicating significant concerns. This comparison necessitates careful interpretation, acknowledging the influence of developer familiarity with Dart. To ensure data accuracy, we automated SonarQube analyses for 15 projects per framework, aiming for a balanced and unbiased evaluation.

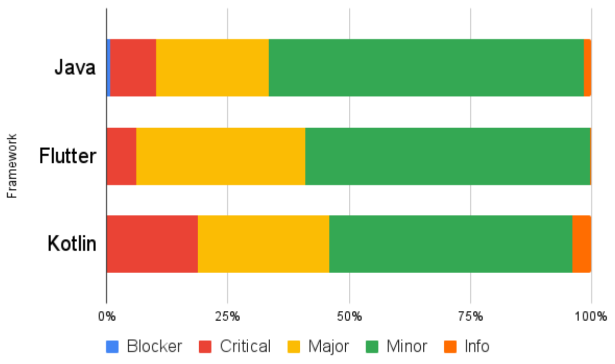
4. Results of Comparative Analysis

The comparative study of the Kanban board app in Kotlin, Dart, and Java, coupled with SonarQube's automated analysis of 45 open-source projects, provided valuable insights into code quality, maintainability, and developer productivity. This section discusses the outcomes of the app development and the SonarQube assessments.

Kotlin and Dart emerge as notably efficient in their coding practices, with Kotlin achieving 1,467 LOC and Dart slightly higher at 1,515 LOC, both outperforming Java's 1,748 LOC. This efficiency reflects not only in reduced complexity but also in enhanced maintainability. The evaluation of code smells further distinguishes these frameworks, with Kotlin and Dart showing a close count of 24 and 22 code smells respectively, indicating a comparable level of code quality. Notably, Dart's codebase stands out for having zero critical issues, underscoring its robustness in security and stability. In contrast, Java's codebase demonstrates a broader array of code smells with 16 critical, 22 major, and 35 minor issues as depicted in Table 1, suggesting a pressing need for stringent quality controls. This concise overview encapsulates the comparative strengths and weaknesses of each framework, highlighting Dart's exceptional performance in mitigating critical issues, and underlines the importance of rigorous code quality management across all platforms.



(Figure 1) Severity Levels by Lines of Code (LOC) in Java, Flutter, and Kotlin Projects



(Figure 2) Normalized Comparison of Severity Levels in Java, Flutter, and Kotlin Projects

Java Projects had a higher total Lines of Code (LOC) at 52,901, with 3,590 code smells. A significant portion of these were Minor code smells (65%), as depicted in Figure 1 and Figure 2. Critical (9.5%) and Major (23.3%) issues suggest potential areas for improvement in application stability and security.

Kotlin Projects displayed a slightly lower total LOC at 48,865, with 647 code smells. This distribution includes a higher percentage of Critical issues (18.9%), as shown in Figure 1 and Figure 2. This indicates that while Kotlin code bases are generally efficient, there are important areas that require attention to mitigate potential vulnerabilities.

Flutter Projects exhibited a significantly larger total LOC at 174,044, with 1,303 code smells. They maintained a favorable distribution of code smells: Minor (58.8%), Major (34.8%), Critical (6.1%), and Info (0.2%). This distribution underscores Flutter's capability to manage larger codebases with a relatively low incidence of critical issues, emphasizing its suitability for complex application development.

5. Discussion

This study incorporated a community-developed SonarQube plugin for Flutter, which was not originally included in the SonarQube suite. This highlights the flexibility of SonarQube to adapt to new frameworks through community contributions, expanding its utility beyond officially supported languages and tools.

Future research should consider incorporating a broader array of GitHub projects to enhance the reliability of the metrics obtained. This expansion could reveal nuanced insights into code quality and maintenance practices across a wider spectrum of development scenarios, potentially identifying trends and exceptions that are not apparent from a limited dataset.

The current study's limitations include potential biases due to the developers' varying familiarity with the programming languages used and differences in platform capabilities that might affect app performance and maintainability. Addressing these issues, future research should explore additional metrics such as runtime efficiency and user experience, and potentially employ machine learning techniques to predict code quality issues. This proactive approach could help developers maintain higher standards of code quality and app performance.

6. Concluding Remarks

Our research explores the landscape of mobile application development, focusing on the transition from Java to Kotlin and the adoption of Flutter cross-platform development. The study involved developing a Kanban board application in Java, Kotlin, and Dart (Flutter), and evaluating code quality across various open-source GitHub projects to understand developers' preferences for programming languages and frameworks.

The finding underscores Kotlin and Dart as efficient alternatives to Java in terms of LOC and maintainability, with Dart demonstrating a notable absence of critical code quality issues. This reflects the advancements these newer technologies offer over traditional Java, highlighting their potential to enhance development practices by offering cleaner, more maintainable code with fewer critical vulnerabilities.

However, it is imperative to approach these results with an understanding of the nuanced nature of software development. The choice between cross-platform and native development approaches is influenced by various factors, including but not limited to application requirements, developer skill sets, and project timelines. While Kotlin and Flutter present compelling advantages, Java continues to be a viable option for certain development contexts due to its established ecosystem and broad developer community.

The research advocates for a balanced approach to framework selection, considering not only the immediate productivity gains but also long-term maintainability and security implications. The emphasis on continuous quality assurance, irrespective of the chosen framework, emerges as a pivotal factor in the success of mobile application projects. Future explorations should aim to broaden the scope of analysis, incorporating additional metrics and larger datasets to validate and extend these findings, thereby enriching the decision-making process for mobile application development further.

References

[1]. Mohamed Abdal Mohsin Masaad Alsaid, “A Comparative Analysis of Mobile Application Development Approaches”, Proceedings of the Pakistan Academy of Sciences: A: Physical and Computational Sciences, pp. 35-45, 2021

[2]. F. Palomba, “On the diffuseness and the impact on maintainability of code smells: a large-scale empirical investigation”, Empirical Software Engineering, pp. 1188-1221, 2017

[3]. F. Palomba, “Toward a smell-aware bug prediction model”, Ieee Transactions on Software Engineering, pp. 194-218, 2019

[4]. Osama M.A. AL-atraqchi, “A Proposed Model for Build a Secure Restful API to Connect between Server Side and Mobile Application Using Laravel Framework with Flutter Toolkits”, cuesj [Internet], 2022

[5]. Péter Hegedűs, “Static code analysis alarms filtering reloaded: a new real-world dataset and its ml-based utilization”, IEEE Access 10, pp. 55090-55101, 2022

[6]. M. Martínez and B. Mateus, “Why did developers migrate android applications from java to kotlin?”, Ieee Transactions on Software Engineering, pp. 4521-4534, 2022

[7]. A. Mazuera-Rozo, C. Escobar-Velásquez, J. Espitia-Acero, D. VegaGuzmán, C. Trubiani, M. Linares-Vásquezet al, “Taxonomy of security weaknesses in java and kotlin, arXiv:2201.11807v1, 2022

[8]. G. Hecht, R. Rouvoy, N. Moha, & L. Duchien, “Detecting antipatterns in android apps”, ACM international conference on mobile software engineering and systems, pp. 148-149, 2015

[9]. M. Lamothe, W. Shang, & T. Chen, “A3: assisting android api migrations using code examples”, Ieee Transactions on Software Engineering, pp. 417-431, 2022

[10]. Ardito, R. Coppola, G. Malnati, & M. Torchiano, “Effectiveness of kotlin vs. java in android app development tasks”, Information and Software Technology, pp. 106374, 2020

[11]. Flauzino, M., Veríssimo, “Are you still smelling it? A comparative study between Java and Kotlin language”, SBCARS, pp. 23-32, 2018

[12]. Anonymous, Firestore Documentation, [Online]. URL: https://firebase.google.com/docs, last visited on April 9

[13]. Anonymous, Flutter Documentation, [Online]. URL: https://docs.flutter.dev, last visited on April 12, 2024

[14]. Anonymous, Bloc State Management Library, [Online]. URL: https://bloclibrary.dev, last visited on April 9, 2024

[15]. Sewak J., MVVM Architecture in Android Using Kotlin, [Online]. URL: https://medium.com/@jecky999/mvvm-architecture-in-android-using-kotlin-a-practical-guide-73f8de1d9c58, last visited on April 8, 2024

[16]. Varotariya V., MVVM Architecture Design Pattern for Android. OneClick IT Consultancy, [Online]. URL: https://oneclickitsolution.com/blog/choose-android-mvvm-over-mvparchitecture, last visited on April 9, 2024

[17]. Anonymous, SonarSource Documentation, [Online]. URL: https://www.sonarsource.com/, last visited on April 9, 2024

[18]. Anonymous, Docker, [Online]. URL: https://docker.com, last visited on April 11, 2024

[19]. Anonymous, Flutter plugin, [Online]. URL: https://github.com/insideapp-oss/sonar-flutter, last visited on April 15, 2024