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

List of Figures

List of Tables

List of Abbreviations

SDK	Software Development Kit
API	Application Programming Interface
LOC	Lines of Code
MVVM	Model-View-ViewModel
NOD	Number of Downloads
NOR	Number of Reviewers
AR	Application Ranking
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 lots of approaches to developing mobile apps 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 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.

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.

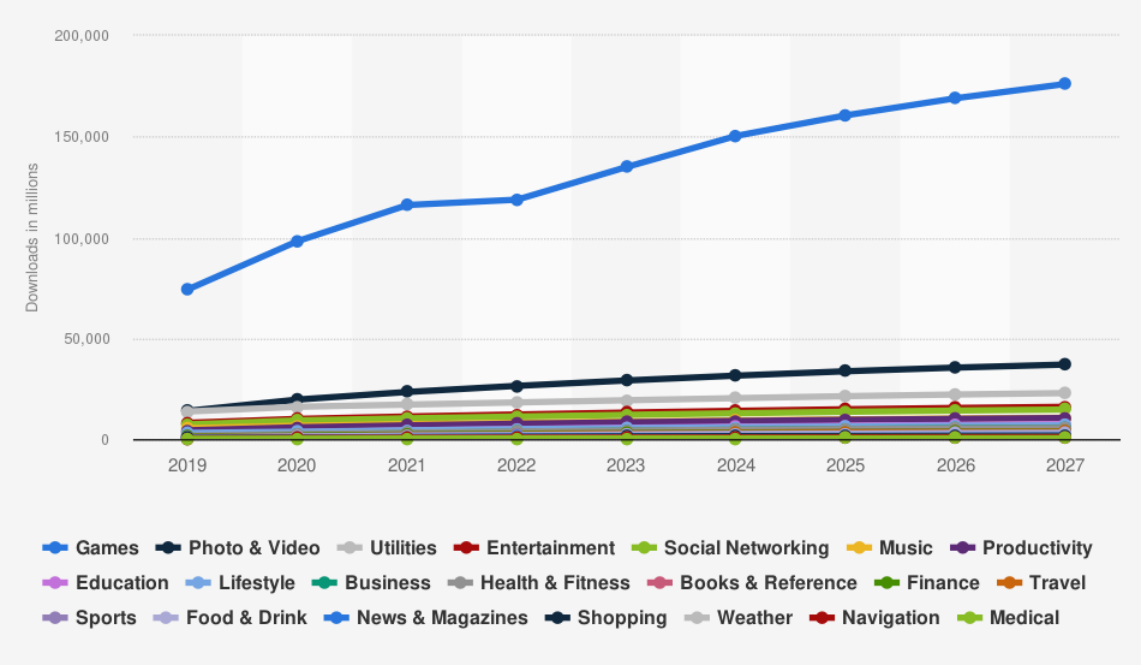

Figure 1.1: Number of mobile app downloads worldwide from 2019 to 2027, by segment(in million downloads) [1]

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 Evolution of Mobile Development Approaches

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 capa-

bilities. 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 Flut-

ter, 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.

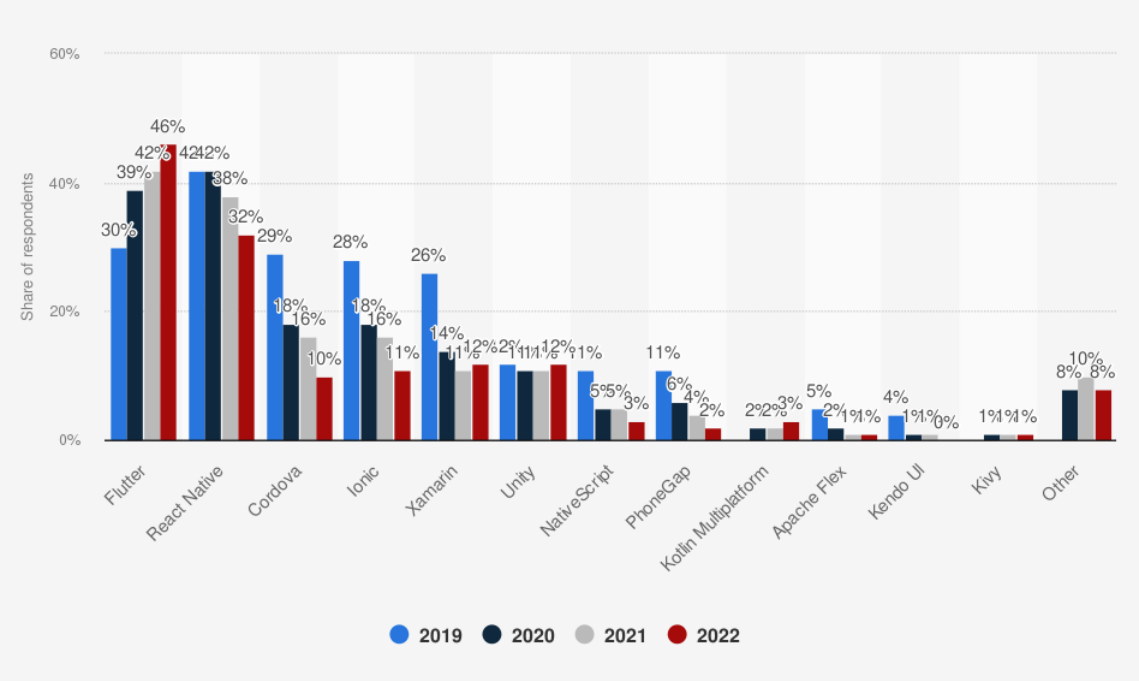

Figure 1.2: Cross-platform mobile frameworks used by software developers worldwide from 2019 to 2022[2]

Objectives

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

2. To analyze the maintainability of Java, Kotlin, and Flutter 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 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 Flutter: 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 Flutter on mobile application main-

tainability? 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 Flutter—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 framework 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 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 developers[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 devel-

opments 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 frameworks. 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 framework'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 framework 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 frameworks, Java, Kotlin, and Dart (Flutter), by developing an identical Kanban board application. The investigation aims to evaluate each framework's development efficiency, usability, and initial code quality. The study provides a comprehensive account of the developmental process followed for each framework, 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 framework 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, we opted for a To-do application to fulfill our research objectives of identifying a framework best suited for application development.

In the next phase of our project development, we meticulously evaluated the features to include and the backend and database to use. After thorough consideration, we concluded that Firebase is the optimal choice for our backend. Firebase's significantly reduced coding effort, thanks to its extensive range of pre-built features that seamlessly

integrate into our project, primarily drove this decision. Moreover, Firebase offers two database options—Realtime and Firestore—and we selected Firestore for its superior performance and scalability.

Once we settled on our database, we embarked on a user-centric approach to feature selection. We identified the features that best serve our project's needs. 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: We utilize the Firebase database to store and manage user data and tasks, enabling real-time updates and synchronization across all devices.

Our next step was to create a detailed design for our application. After some consideration, we used the Figma application because of 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, we recognized the importance of avoiding improvisation and thus decided to prioritize the design phase before diving into coding.

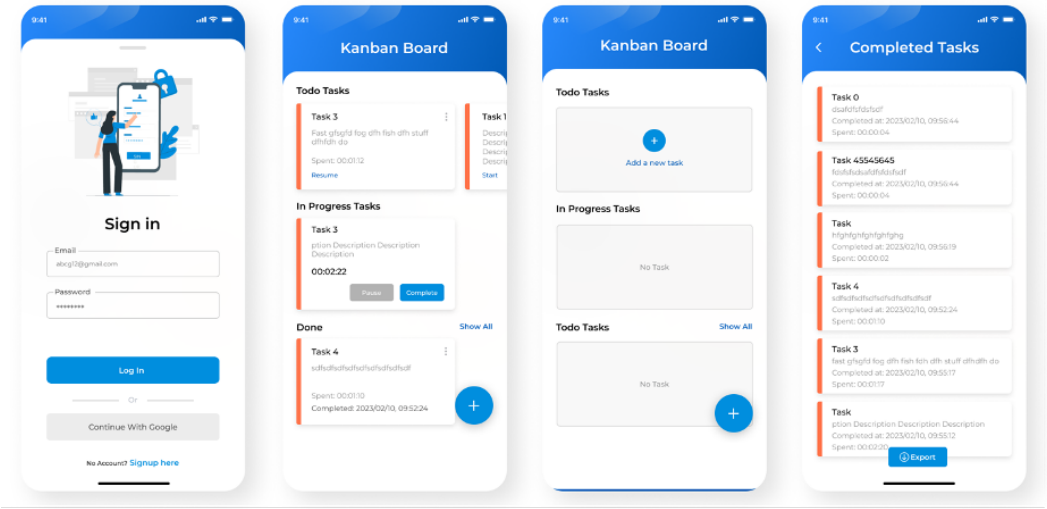

Figure 3.1: Planned view of Kanban board application. Designed by Figma

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

3.2.2 Flutter Application Development

We decided to begin with the Flutter application to ensure a smooth development process due to our familiarity with the framework. Our feature specifications and Figma designs were already in place, which made the process easier. 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,

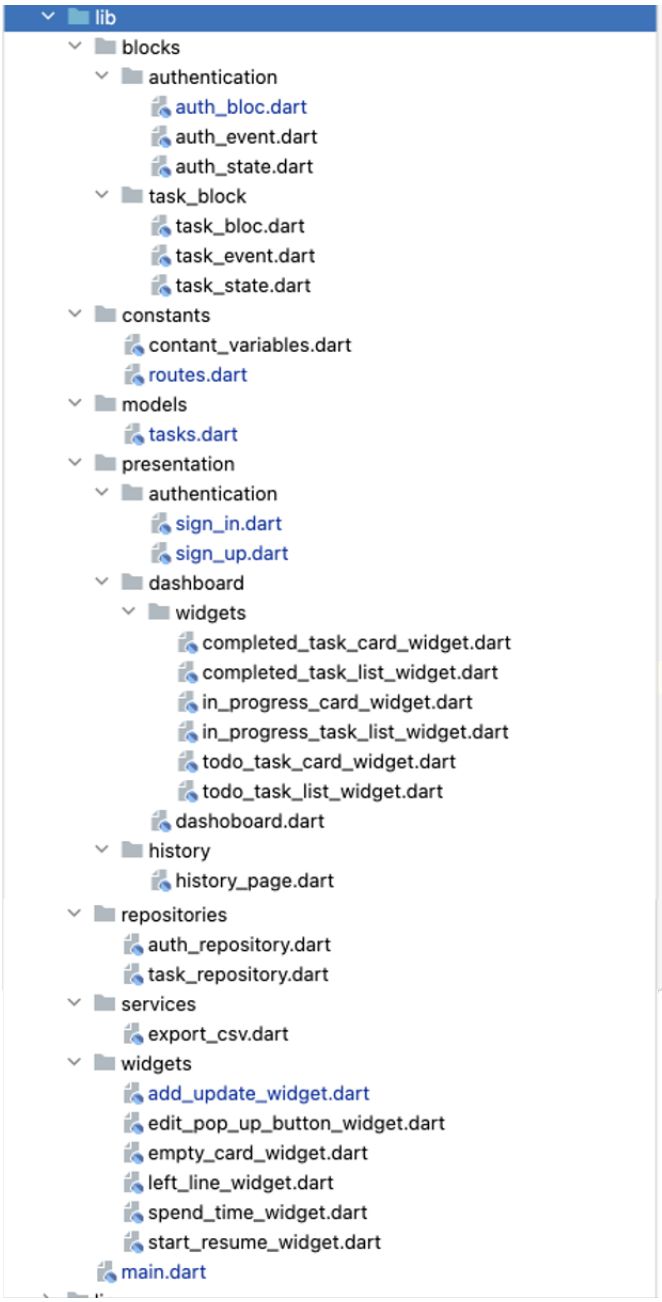

Figure 3.2: Flutter version of the application's project structuring

and 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.

It took us around 20 hours to complete the project, and the total number of files was 29. 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, we tried to follow the clean architecture mentioned in the book [48] by Robert Cecil Martin (also known as Uncle Bob).

3.2.3 Kotlin Application Development

We developed a Kotlin-based Kanban board application after completing our initial Flutter project. Following our successful Flutter project, we adopted Kotlin as our first choice for Android Native development. Our Kotlin project was designed to support a Kanban board application and consisted of 26 .kt files and 30 .xml files. The development process took approximately 24 hours, primarily due to my limited familiarity with the Kotlin Application Development. However, despite these challenges, we leveraged the

Model-View-ViewModel (MVVM) architecture to enhance maintainability and facilitate a clear separation of concerns. The MVVM [49] architecture enabled us 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.

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

As it is shown in Figure 3.3, we organized the Kotlin code 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, we focused on developing the Android application's Java version. The project structure had to remain consistent with the Kotlin version, and to achieve this, we implemented advanced decompiling tools that facilitated the conversion of Kotlin code into Java. This approach significantly streamlined our 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, we saved considerable development time and effort, allowing us to 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

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

hours.

The Figure 3.4 indicates that the resultant Java project was structured similarly 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 sub-packages 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.

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. [Online; accessed 19-October-2023].

[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 16-July-2022].

[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?, 2022. URL: https://www.iadisportal.org/digital-library/flutter-the-future-of-application-development.

[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] URL: https://bloclibrary.dev.

[46] URL: https://chornthorn.github.io/getx-docs/.

[47] URL: https://pub.dev/packages/provider.

[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, Jun 2023. URL: https://medium.com/@jecky999/mvvm-architecture-in-android-using-kotlin-a-practical-guide-73f8de1d9c58.

[50] Tim Krieger and Daniel Meierrieks. Terrorist financing and money laundering. Working Papers CIE 40, Paderborn University, CIE Center for International Economics, 2011. URL: https://EconPapers.repec.org/RePEc:pdn:ciepap:40.

[51] John. Roth, Douglas. Greenburg, Serena. Wille, and National Commission on Terrorist Attacks upon the United States. *Monograph on Terrorist Financing*. National Commission on Terrorist Attacks upon the United States, 2004. URL: https://nla.gov.au/nla.cat-vn3838118.

[52] FATF. Terrorist financing. *Financial Action Task Force*, February 2008.

[53] Michael Freeman and Moyara Ruehsen. Terrorism financing methods: An overview. *Perspectives on Terrorism*, 7(4):5–26, 2013. URL: http://www.jstor.org/stable/26296981.

[54] Winston Maxwell, Astrid Bertrand, and Xavier Vamparys. Are AI-based Anti-Money Laundering (AML) Systems Compatible with European Fundamental Rights? In *ICML 2020 Law and Machine Learning Workshop*, Vienne, Austria, July 2020. Conférence en ligne. URL: https://hal.science/hal-02884824.

[55] Lei Cai, Zhengzhang Chen, Chen Luo, Jiaping Gui, Jingchao Ni, Ding Li, and Haifeng Chen. Structural temporal graph neural networks for anomaly detection in dynamic graphs, 2020.

[56] Kaize Ding, Jundong Li, Rohit Bhanushali, and Huan Liu. *Deep Anomaly Detection*

on Attributed Networks, pages 594–602. 2019. URL: https://epubs.siam.org/doi/abs/10.1137/1.9781611975673.67.

Appendices

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 방법을 선택하는 데 어려움을 겪고 있습니다. 이 연구는 주로 Java에서 Kotlin으로의 선호도 변화와 Flutter의 사용 증가에 중점을 두고, 네이티브 및 크로스플랫폼 애플리케이션 개발 접근 방식을 비교합니다. 이 연구는 Java, Kotlin, Dart(Flutter)를 사용하여 동일한 애플리케이션을 생성함으로써 모바일 애플리케이션 개발에서 개발자의 프로그래밍 언어 및 프레임워크 선택에 영향을 미치는 요인에 대한 실용적인 통찰력을 제공합니다. 또한, 이 연구는 45개의 오픈소스 GitHub 저장소에서 코드 품질을 검토함으로써 개발의 모범 사례를 탐구합니다. 연구는 반자동 SonarQube 평가를 사용하여 LOC 및 코드 스멜을 평가함으로써 특정 언어나 프레임워크 선택이 코드 유지 관리 및 개발 효율성에 미치는 영향을 결정합니다. 예비 결과는 두 접근 방식에서 생성된 코드 품질의 차이를 보여주며, 개발자가 코드 스멜을 줄이고 프로젝트 유지 관리를 개선하기 위해 언어와 프레임워크 선택을 최적화하는 데 도움이 되는 정보를 제공합니다.