Bulletin of Business and Economics, 13(1), 355-362
https://bbejournal.com
https://doi.org/10.61506/01.00211

An Automated Performance Enhancement Approach for Mobile Application

**Muhammad Tahir[1], Muhammad Rahim Zafar[2], Muhammad Talha Bashir[3],
Saleem Zubair[4], Muhammad Waseem Iqbal[5], Fawad Nasim[6]**

Abstract

In the rapidly evolving landscape of mobile applications, the demand for high-quality, performance-driven software is paramount. However, the swift pace of development often leads to the introduction of code smells—bad programming practices that compromise both code quality and application performance. These code smells, if left unaddressed, can result in increased memory consumption, energy consumption, and CPU utilization, ultimately leading to a suboptimal user experience. This paper presents an automated approach for the detection and refactoring of code smells in Android applications, with a focus on improving performance. The proposed approach involves the development of a plugin integrated with Android Studio, which employs static code analysis to identify code smells. The plugin encompasses a customizable rule-based framework that allows for the detection of various code smells unique to Android development. To validate the approach, a comprehensive experiment is conducted. The experiment assesses the effectiveness of the proposed method in detecting code smells and explores the impact of refactoring on application performance. The results showcase that the proposed plugin successfully detects code smells in various open-source Android applications. Moreover, the integration of refactoring recommendations significantly improves the performance of the applications, as demonstrated through memory, energy, and CPU consumption metrics. Comparison with existing tools reveals that the proposed approach offers superior performance in terms of both code smell detection and refactoring. Additionally, the approach bridges the gap left by some existing tools by identifying previously undetected code smells, such as "string concatenation." The presented method not only enhances code quality but also contributes to the overall performance optimization of Android applications. As mobile applications continue to play an increasingly central role in modern life, the importance of maintaining high-quality code that performs optimally cannot be understated. This work provides a valuable contribution towards achieving these goals, offering developers a powerful tool for ensuring that their applications not only meet but exceed user expectations in terms of quality and performance.

Keywords: mobile applications, code smells, performance optimization, Android Studio plugin, static code analysis, refactoring, application performance, memory consumption, software engineering

1. Introduction

The rapid growth of mobile application development has significant implications for quality attributes (Usman, Iqbal, & Khan, 2020). However, in their pursuit of capturing the business market, developers often make compromises on design patterns and coding styles, potentially leading to performance degradation (Rasool & Ali, 2020). Despite the increasing demand for applications, compromising quality and performance remains unacceptable (Usman, Iqbal, & Khan, 2020). Mobile applications have become integral to everyday life (Usman, Iqbal, & Khan, 2020). Over the years, mobile app development has surged, predominantly centered on the Android and Apple iOS platforms (Rasool & Ali, 2020). However, developers frequently prioritize functional needs at the expense of architectural considerations, resulting in subsequent quality issues (Usman, Iqbal, & Khan, 2020).

Testing plays a pivotal role in the mobile application development process, with performance testing being a critical component (Nguyen, Huynh, & Nguyen, 2016). Unfortunately, performance testing is often deferred until the later stages of development, impacting the application's performance (Nguyen, Huynh, & Nguyen, 2016). As mobile applications operate on devices with limited battery life, improper development can lead to performance issues known as "design smells" (Usman, Iqbal, & Khan, 2020). These design smells stem from various sources, including bad design practices and disregarding green development principles (Rasool & Ali, 2020). They manifest as memory, energy, and time inefficiencies (Khan, Lee, Abbas, Abbas, & Bashir, 2021). Resource leaks, including memory, CPU, and energy consumption issues, are common code smells (Boutaib, Bechikh, Palomba, Elarbi, Makhlouf, & ben Said, 2021). Even string handling practices, such as concatenation and string builder usage, can adversely affect performance (Zhang & Elbaum, 2012).

Addressing these code smells is crucial for enhancing performance (Institute of Electrical and Electronics Engineers & International Conference on Software Engineering, 2013). However, manual detection of code smells is challenging for large applications (Kong, Li, Gao, Liu, Bissyandé, & Klein, 2019). Automated methods for detection and refactoring have emerged (Kim, ACM Digital Library, & Association for Computing Machinery Special Interest Group on Knowledge Discovery & Data Mining, 2009), but they may not cover all smell types (Rodriguez, Mateos, & Zunino, 2017). Researchers aim to categorize code smells (Kim, 2017), refactor identified smells (Nguyen, Huynh, & Nguyen, 2016), apply these approaches to open-source applications (Kong, Li, Gao, Liu, Bissyandé, & Klein, 2019), and compare different tools' effectiveness (Khan, 2016; Skretting & Gronli, 2020).

1.1. Objectives

This research strives to contribute to mobile application performance enhancement by introducing an automated method for detecting and refactoring performance-degrading code smells (Das, di Penta, & Malavolta, 2020). The ultimate goal is to improve user

[1] Department of Software Engineering, Superior University Lahore, Pakistan, tahirasml1@gmail.com
[2] Department of Software Engineering, Superior University Lahore, Pakistan, rahmzafar123@gmail.com
[3] Master of Science (MSc) in Data Analytics, Dublin Business School, Ireland, talha.lqp@gmail.com
[4] Department of Computer Science, Superior University Lahore, Pakistan, saleem.zubair@superior.edu.pk
[5] Associate Professor, Department of Software Engineering, Superior University Lahore, Pakistan, waseem.iqbal@superior.edu.pk
[6] Department of Software Engineering, Superior University Lahore, Pakistan, fawad.nasim@superior.edu.pk

satisfaction and gain a competitive advantage in the dynamic mobile app market (Boutaib, Bechikh, Palomba, Elarbi, Makhlouf, & ben Said, 2021).

1.2. Research Questions

To address the stated objectives, the following research questions will guide our study
RQ1: What are the distinct code smells that have an impact on the performance of mobile applications?
RQ2: Can a comprehensive taxonomy of performance-degrading code smells be established, complete with detection rules?
RQ3: How does the proposed approach perform in detecting code smells?
RQ4: What is the effectiveness of refactoring the identified "String Concatenation" smell using an automated approach?
RQ5: How does the proposed approach compare to existing methods in terms of its impact on enhancing performance?
By answering these research questions, we intend to contribute to the existing
knowledge on mobile application performance improvement and the mitigation of performance-degrading code smells.

1.3. Motivation

This research is motivated by the significant impact of rapid mobile application development on quality attributes. Code smells have been identified as a critical factor contributing to performance degradation, and this study aims to address these issues. By detecting and refactoring these code smells, we strive to enhance mobile application performance and overall quality.

1.4. Problem Statement

While various code smells impacting performance have been identified in literature, some remain undetected and may not be resolved by existing methods. The detection and refactoring of these code smells are crucial to improve mobile application performance. Furthermore, a comparative analysis of existing tools in detecting and refactoring code smells is needed to identify their limitations and propose more effective solutions.

1.5. Scope of the Study

This study focuses on performance-degrading code smells in mobile applications. The research emphasizes the detection and refactoring of the "String Concatenation" smell due to its significant impact on performance. The study aims to develop an automated approach that enhances the performance of mobile applications by addressing these code smells.

2. Literature Review

Code smells are coding practices that can negatively affect Android app performance. These practices can lead to issues like increased memory usage, higher power consumption, and slower responsiveness. Researchers have extensively studied the impact of code smells on Android app performance and proposed approaches to detect and fix these issues.
Researchers use static analysis tools and machine learning techniques to automatically identify code smells. Static analysis tools like Android Lint examine source code for inefficiencies and potential problems (Das, di Penta, & Malavolta, 2020). Machine learning methods use patterns and metrics to identify problematic code segments (Degu, n.d.).
Code refactoring involves restructuring code to eliminate code smells and improve performance. Automated tools like aDoctor and HOT-PEPPER help developers detect and refactor code smells, providing suggestions to enhance code quality and efficiency (Rodriguez, Mateos, & Zunino, 2017) (Kong, Li, Gao, Liu, Bissyandé, & Klein, 2019).
Efficient energy consumption is essential for mobile applications. Code smells can contribute to higher energy usage. Tools like HOT-PEPPER focus on identifying energy-related code smells and patterns, enabling developers to create more energy-efficient applications (Kong, Li, Gao, Liu, Bissyandé, & Klein, 2019).
Empirical studies evaluate the effectiveness of code smell detection and refactoring techniques. These studies assess memory usage, energy consumption, and overall application performance (Kong, Li, Gao, Liu, Bissyandé, & Klein, 2019). Challenges include false positives and limitations of automated tools (Institute of Electrical and Electronics Engineers & International Conference on Software Engineering, 2013).
Future research could refine code smell detection algorithms, enhance tool accuracy, and explore dynamic analysis methods. Addressing these challenges could lead to improved code quality and better Android application performance (Pldi 12 Proceedings Committee, 2013)

2.1. Taxonomy of Mobile Specific Code Smells

In the realm of mobile application development, a taxonomy of Android-specific code smells has emerged from prior literature. These code smells significantly impact application performance and have been grouped into a comprehensive taxonomy. This classification is divided into five principal categories.

2.1.1. Resource Leaks

Resource leaks transpire when acquired resources are not properly released by the program, leading to issues such as memory, CPU, or energy leaks. This deficiency in resource management precipitates performance degradation. Resource leaks encompass a spectrum of types, including memory leaks, CPU consumption, and energy leaks. Automated detection of these code smells using bytecode or static code analyzers is pivotal for enhancing performance. The literature offers an array of tools that assist in detecting resource-related bugs; however, rectifying or eradicating these issues remains a challenge (Palomba, di Nucci, Panichella, Zaidman, & de Lucia, 2017).

2.1.2. Database Smells

Database-related code smells encompass issues in database interactions that can culminate in suboptimal data management, undermining application responsiveness. These code smells can impede the application's seamless functioning, particularly when dealing with extensive data operations.

Table 1: Mobile Application Code Smells

ID	Smell Name	Smell Description	Smell Context+ Affect	Detection Rule
S1	Leaking Inner Class	This smell arises when non-static inner classes hold references of outer classes, causing memory leak issues.	Implementation context, Memory Consumption	(Class A (outer) is concrete) && (Class B(inner)) && (Class B holds reference of Class A)
S2	No Low memory resolver	A lack of the "onLowMemory" method can lead to memory and stability issues.	Implementation context, Memory issue and stability issue	(A concrete class A with activity class) && (Class A with no method of "onLowMemory")
S3	Leaking Thread	Starting threads without proper termination can result in memory leaks.	Implementation context, Memory leaks	(Class A is concrete and uses thread) && (Class A started thread with no method of Thread.stop())
S4	Static View	Usage of static fields of Views leads to memory leaks and inefficient performance.	Memory, Efficiency, Performance	Look for static ButtonsViews, e.g., Button, View, or any layout.
S5	Static Bitmap	Utilizing static bitmaps can consume excessive memory resources.	Memory, Performance	Look for static bitmaps in applications.
S6	Member Ignoring Method	Non-static methods not utilizing class attributes cause performance issues.	Efficiency, Performance	(Method M () in Class A) && (M() is not using any attribute of Class A)
S7	HashMap Usage	Excessive use of HashMap can lead to implementation and performance issues.	Implementation, Performance	Use of HashMap<>
S8	Use of Getter & Setter	Excessive use of getter and setter methods can impact implementation and performance.	Implementation, Performance	Get and set method in class
S9	Wake Lock	Improper usage of wake locks for resources can affect UI implementation and energy consumption.	UI implementation, Memory, Energy consumption	(Class is using wakelock()) && (Time frame is not mentioned)
S10	Unclose Closeable	Implementing Closeable interfaces without calling the close method leads to memory issues.	Implementation, Memory	(Class A implements the interface of closeable) && (missing method of Close() of closeable in Class)
S12	String Concatenation	Using string concatenation instead of string builder affects energy consumption and processing time.	Implementation, Energy + processing time	Use of string instead of string builder/ string buffer. Try not to use the "+" operator to concatenate the string.

2.1.3. UI Smells

User Interface (UI) code smells encompass design-related issues that adversely affect user experience and overall application performance. These anomalies can manifest in user interface components, potentially leading to slow responsiveness and user dissatisfaction. Network Smells: Network-specific code smells encompass concerns related to network operations. These issues can impact the application's network usage efficiency and responsiveness.

2.1.4. Others

This category encapsulates code smells that do not fit squarely within the aforementioned classifications yet still contribute to performance deterioration. These code anomalies may encompass miscellaneous implementation issues that compromise overall application performance.

2.2. Mobile Application Code Smells

In the mobile application domain, various code smells have been identified that directly impact performance. These code smells exhibit specific characteristics that facilitate their identification and understanding. Notably, different tools are available for detecting distinct code smells, offering developers the means to enhance application performance.
Numerous tools and approaches have been employed to detect these code smells:

Table 2: Android Code Smells

Ref	Detected Smells	Tool Used	Type of Approach
(Palomba, di Nucci, Panichella, Zaidman, & de Lucia, 2017)	S1, S2, S3, S6, S7, S8, S9, S10, S13, S14, S24, S25, S26	aDoctor	Static code analysis
(Gattal, Hammache, Bousbia, & Henniche, 2021)	S21, S13	Randroid	Static code analysis
(Bhatt & Furia, 2020)	Energy and memory leaks	Plumb Droid	Byte code analyzer
(Hecht, Benomar, Rouvoy, Moha, & Duchien, 2016)	S1, S2, S6, S8, S19, S21, S11, S5, S7	Paprika	Byte code analyzer

In summation, the taxonomy of mobile-specific code smells in Android applications categorizes these anomalies into various groups. These code smells, such as resource leaks, database issues, UI anomalies, network problems, and others, impact application performance. Within these categories, distinct code smells have been identified and documented, often with specific detection rules and contexts. Various tools and approaches have been employed to detect these code smells, contributing to the optimization of mobile application performance.

3. Methodology

To address the challenge of detecting code smells within large Android applications, we have devised a comprehensive plugin that seamlessly integrates with Android Studio. This plugin serves as a robust tool for identifying code smells within Android applications, aiding in their detection and subsequent resolution. The methodology primarily relies on static code analysis techniques. Our approach involves multiple key components, as outlined below:

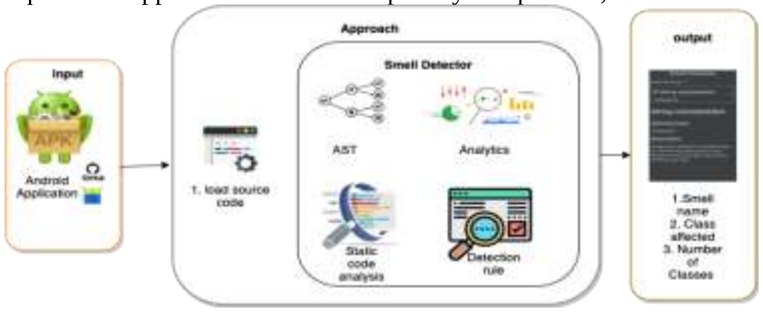

Figure 1: Proposed Approach.: Source F. Palomba et al., 2020

3.1. Examination of Android Applications

The process commences with the comprehensive analysis of Android applications. This entails programmatically scrutinizing the application's source code and XML files, aiming to identify instances of code smells.

3.2. Recognizing Rule Violations

The plugin is equipped with a set of predefined rules corresponding to various code smells. These rules are crafted to identify instances within the source code where violations of coding standards are apparent. Such violations serve as indicators of potential code smells.

3.3. Providing Smell Details

Upon identifying a code smell, the plugin provides detailed insights, including the number of instances of the detected smell and their specific locations within the codebase.

3.4. Static Code Analysis

The core of our approach relies on static code analysis. This involves analyzing the Android application's source code using the AST (Abstract Syntax Tree) mechanism. The source code, consisting of both Java and XML files, is loaded into Android Studio, where it undergoes preprocessing.

3.5. AST Process

The AST process classifies code elements into distinct categories, including class instance creation, method declaration, method invocation, and variable declaration. This categorization is instrumental in facilitating the subsequent detection process.

3.6. Smell Detection Rule Application

Following AST processing, the plugin applies the predefined smell detection rules. These rules are devised to identify code fragments that violate established coding standards. When a code element fails to meet the criteria set by a specific rule, it indicates the presence of a code smell.

3.7. Refactoring and Suggestions

Upon detecting code smells, the plugin not only identifies the issues but also provides refactoring suggestions. These suggestions aid developers in rectifying the identified smells by offering code improvements. With a simple click, the suggested refactoring changes can be incorporated into the codebase.

3.8. Automation and Efficiency

The plugin's automated nature streamlines the detection and resolution of code smells. Manual tracking and rectification processes, which are prone to human errors and time-consuming, are minimized. The plugin ensures that all instances of a detected smell are accurately identified and provides suggestions for seamless refactoring.

3.9. User Feedback and Visualization

The plugin presents detected code smells to the user via a GUI (Graphical User Interface) dialog box. This interface provides comprehensive details about the identified smells, enabling users to understand the issues at hand.

3.10. Tool's Architecture

The proposed plugin is built using the IntelliJ SDK and is tailored for Android Studio. It operates within a two-layer architecture

3.10.1. Presentation Layer

This layer encompasses essential controls and GUI components. A Java Swing-based GUI dialog box effectively displays the detected code smells and their associated details. The CoreDriver class contains the core logic for the dialog, governing its various controls, such as Start Dialog, No Smell, Abort, and Smell List.

3.10.2. Application Layer / Business Logic

The business logic layer is responsible for executing the core functionalities of the plugin:

- Analyzer: This component carries out the detection of code smells, identifying instances of rule violations
- Analytics: It offers statistical analysis of detected code smells based on user-selected preferences, facilitating further investigation.
- AST (Abstract Syntax Tree): This module scrutinizes the elements of the source code and applies the predetermined rules for smell detection.

4. Experiment

The experiment aimed to assess the effectiveness of the proposed approach in detecting and refactoring code smells, as well as to evaluate the impact of refactored smells on performance. The experiment was conducted using the following environment:

Android Studio Bumblebee | 2021.1.1 patch 3 , Android Gradle Plugin version 7.1.3,Gradle version 7.2, Android JDK version 11.0.11, Android Virtual Device (AVD) Nexus 5X API 27 (Portrait)

The experiment utilized an Android application named "TestAndroid" for detailed testing and validation. The following features were evaluated:

- Performance comparison between using `String` and `StringBuilder` in Android application.
- Correctness and effectiveness of the proposed approach in detecting code smells.
- Comparison of performance between using `String` and `StringBuilder`.

4.1. Test Execution

Test cases were designed to evaluate the performance of using `String` vs. `StringBuilder`. Different input string numbers and lengths were utilized to measure energy, memory, and CPU consumption. The results were recorded for further analysis.

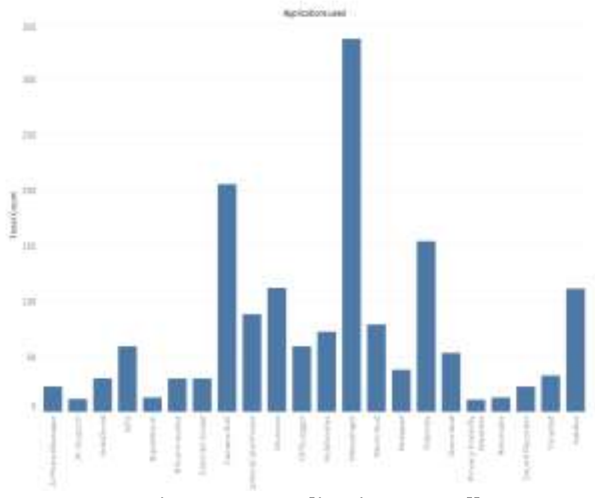

Figure 2: Application's Smells

4.2. Test Results

The experiment showed that using `StringBuilder` significantly outperformed using `String` concatenation. The execution time for `String` concatenation was much higher compared to `StringBuilder`. The complexity of `StringBuilder` operations remained linear (O(n)), whereas `String` concatenation had quadratic complexity (O(n^2)). Performance parameters were visually represented through graphs showing CPU consumption, energy consumption, and memory consumption. These graphs demonstrated that using `StringBuilder` resulted in lower CPU, energy, and memory consumption compared to `String` concatenation.

4.2.1. Smell Detection and Refactoring

The proposed plugin was evaluated on 22 open-source Android applications from GitHub and F-Droid. The plugin detected and refactored code smells in these applications. It detected 17 different smells and performed refactoring on 7 of them. The most frequently detected smells were MIM (Member Ignoring Method), NLMR (No Low Memory Resolver), SC (String Concatenation), LIC (Leaking Inner Class), LT (Leaking Thread), and DW (Durable Wakelock).

4.2.2. Comparison with Other Tools

The proposed plugin was compared with existing tools such as xAL and PAPRIKA. The comparison involved precision and recall metrics. Our plugin demonstrated higher true positive (TP) results, resulting in average precision and recall values of 0.95 and 0.97, respectively. This suggests that the proposed approach is effective in detecting code smells and achieving better performance.

4.2.3. Visualization and Conclusion

The results were visualized through stacked bar charts and performance graphs, providing a comprehensive understanding of the experiment's outcomes. The proposed approach demonstrated its effectiveness in detecting and refactoring code smells, leading to improved application performance. In conclusion, the experiment successfully demonstrated the effectiveness of the proposed approach in detecting code smells and enhancing performance. The approach's performance was compared favorably with existing tools, indicating its superiority in terms of precision and recall. The plugin's capabilities were validated across a range of open-source Android applications, affirming its value in real-world scenarios.

5. Conclusion

In the modern era, mobile phones have become integral to daily life, replacing computers in various work and entertainment activities. The rapid growth in mobile usage has led to an increasing demand for applications that provide quality and performance. However, developers often overlook fundamental programming practices during the rapid development process, leading to the introduction of code smells and performance degradation. This necessitates the need for tools that can enhance both quality and performance. Code smells, which represent bad programming practices, can have a detrimental impact on application performance. Addressing these smells is crucial for maintaining good performance, as they directly affect memory usage, power consumption, and CPU utilization. Mobile applications heavily rely on performance, making it a critical aspect of their quality. This study identified the gaps in the existing literature regarding the detection and refactoring of code smells in Android applications. Certain code smells, such as "string concatenation" and "static views," were not adequately covered. Furthermore, many code smells lacked comprehensive refactoring solutions. Additionally, the study aimed to investigate the empirical effects of code smells on mobile application performance. The proposed automated approach for code smell detection and refactoring in mobile applications demonstrated its validity and effectiveness through experiments. The approach accurately detected code smells, associated them with instance names and the number of instances detected, and provided refactoring suggestions. Experimental results showcased improved processing time through the elimination of string concatenation. Evaluating 22 open-source mobile applications demonstrated the approach's efficacy in detecting code smells. Furthermore, a comparison with existing tools highlighted the superior detection and refactoring performance of the proposed plugin.

5.1. Limitations

While this study provided substantial advancements in code smell detection and refactoring, it still has limitations. The refactoring aspect for some smells is not comprehensive and requires further development. Previous studies also struggled to provide effective refactoring, underscoring the complexity of the task. Additionally, multi-class refactoring for certain smells remains a challenge.

5.2. Future Work

Future work will focus on enhancing the refactoring suggestions to encompass a wider range of smells. Incorporating various types of smells, including network, UI, and database-related smells, will provide a more comprehensive perspective on software quality improvement. The ultimate goal is to create a holistic approach that addresses multiple aspects of software quality and performance, ensuring that mobile applications meet the ever-growing demands of users and the market.

References

Alkandari, M. A., Kelkawi, A., & Elish, M. O. (2021). An Empirical Investigation on the Effect of Code Smells on Resource Usage of Android Mobile Applications. *IEEE Access*, 9,61853-61863.

Amalfitano, D., Riccio, V., Tramontana, P., & Fasolino, A. R. (2020). Do Memories Haunt You? An Automated Black Box Testing Approach for Detecting Memory Leaks in Android Apps. IEEE Access, 8, 12217-12231.

Bhatt, B. N., & Furia, C. A. (2020). Automated Repair of Resource Leaks in Android Applications.

Boutaib, S., Bechikh, S., Palomba, F., Elarbi, M., Makhlouf, M., & ben Said, L. (2021). Code smell detection and identification in imbalanced environments. *Expert Systems with Applications*, 166.

Boutaib, S., Bechikh, S., Palomba, F., Elarbi, M., Makhlouf, M., & ben Said, L. (2021). Code smell detection and identification in imbalanced environments. Expert Systems with Applications, 166.

Boutaib, S., Bechikh, S., Palomba, F., Elarbi, M., Makhlouf, M., & ben Said, L. (2021). Code smell detection and identification in imbalanced environments. *Expert Systems with Applications*, 166.

Carette, A., Ait Younes, M. A., Hecht, G., Moha, N., & Rouvoy, R. (2017). Investigating the Energy Impact of Android Smells.

Chouchane, M., Soui, M., & Ghedira, K. (2021). The impact of the code smells of the presentation layer on the diffuseness of aesthetic defects of Android apps. *Automated Software Engineering*, 28(2).

Chouchane, M., Soui, M., & Ghedira, K. (2021). The impact of the code smells of the presentation layer on the diffuseness of aesthetic defects of Android apps. *Automated Software Engineering*, 28(2).

Couto, M., Saraiva, J., & Fernandes, J. P. (n.d.). Energy Refactorings for Android in the Large and in the Wild.

Das, T., di Penta, M., & Malavolta, I. (2020). Characterizing the evolution of statically-detectable performance issues of Android apps. *Empirical Software Engineering*, 25(4), 2748-2808.

Das, T., di Penta, M., & Malavolta, I. (2020). Characterizing the evolution of statically-detectable performance issues of Android apps. Empirical Software Engineering, 25(4), 2748-2808.

Degu, A. (n.d.). Android Application Memory and Energy Performance: Systematic Literature Review, 21(3), 20-32.

Falessi, D., & Kazman, R. (2021). Worst smells and their worst reasons. In 2021 IEEE/ACM International Conference on Technical Debt (TechDebt), (pp. 45-54). IEEE.

Fatima, I., Anwar, H., Pfahl, D., & Qamar, U. (2020). Detection and Correction of Android-specific Code Smells and Energy Bugs: An Android Lint Extension.

Fatima, I., Anwar, H., Pfahl, D., & Qamar, U. (2020). Tool support for green android development: A systematic mapping study. In ICSOFT 2020 - Proceedings of the 15th International Conference on Software Technologies, (pp. 409-417).

Gattal, A., Hammache, A., Bousbia, N., & Henniche, A. N. (2021). Exploiting the progress of OO refactoring tools with Android code smells: RAndroid, a plugin for Android studio. In Proceedings of the ACM Symposium on Applied Computing, (pp. 1580-1583).

Habchi, S., Moha, N., & Rouvoy, R. (2020). Android Code Smells: From Introduction to Refactoring.

Habchi, S., Moha, N., & Rouvoy, R. (2020). Android Code Smells: From Introduction to Refactoring.

Habchi, S., Moha, N., & Rouvoy, R. (2020). Android Code Smells: From Introduction to Refactoring.

Hamdi, O., Ouni, A., Cinnéide, M. O., & Mkaouer, M. W. (2021). A longitudinal study of the impact of refactoring in android applications. *Information and Software Technology*, 140.

Hecht, G., Benomar, O., Rouvoy, R., Moha, N., & Duchien, L. (2016). Tracking the software quality of android applications along their evolution. In Proceedings - 2015 30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, (pp. 236-247).

Hecht, G., Benomar, O., Rouvoy, R., Moha, N., & Duchien, L. (2016). Tracking the software quality of android applications along their evolution. In Proceedings - 2015 30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, (pp. 236-247).

Hort, M., Kechagia, M., Sarro, F., & Harman, M. (2021). A Survey of Performance Optimization for Mobile Applications. *IEEE Transactions on Software Engineering*.

Hort, M., Kechagia, M., Sarro, F., & Harman, M. (2021). A Survey of Performance Optimization for Mobile Applications. IEEE Transactions on Software Engineering.

Iannone, E., Pecorelli, F., di Nucci, D., Palomba, F., & de Lucia, A. (2020). Refactoring android-specific energy smells: A plugin for android studio. In IEEE International Conference on Program Comprehension, (pp. 451-455).

Institute of Electrical and Electronics Engineers, & International Conference on Software Engineering (35th : 2013 : San Kim, D. K. (2017). Towards Performance-Enhancing Programming for Android Application Development. *International Journal of Contents*, 13(4).

Khan, M. N. (2016). Institutional Governance, Population Dynamics, and Economic Growth: Insights from a Global Panel. *Journal of Business and Economic Options*, *3*(2), 33-45.

Khan, M. U., Lee, S. U. J., Abbas, S., Abbas, A., & Bashir, A. K. (2021). Detecting Wake Lock Leaks in Android Apps Using Machine Learning. IEEE Access, 9, 125753-125767.

Kim, D. K. (2017). Towards Performance-Enhancing Programming for Android Application Development. *International Journal of Contents*, 13(4).

Kim, W., ACM Digital Library, & Association for Computing Machinery Special Interest Group on Knowledge Discovery & Data Mining. (2009). Proceedings of the 3rd International Conference on Ubiquitous Information Management and Communication. ACM.

Kong, P., Li, L., Gao, J., Liu, K., Bissyandé, T. F., & Klein, J. (2019). Automated testing of Android apps: A systematic literature review. *IEEE Transactions on Reliability*, 68(1),45-66.

Kong, P., Li, L., Gao, J., Liu, K., Bissyandé, T. F., & Klein, J. (2019). Automated testing of Android apps: A systematic literature review. *IEEE Transactions on Reliability*, 68(1),45-66.

Kong, P., Li, L., Gao, J., Liu, K., Bissyandé, T. F., & Klein, J. (2019). Automated testing of Android apps: A systematic literature review. *IEEE Transactions on Reliability*, 68(1), 45-66.

le Goaer, O. (2020). Enforcing Green Code with Android Lint. In Proceedings - 2020 35th IEEE/ACM International Conference on Automated Software Engineering Workshops, ASEW 2020, (pp. 85-90).

Mao, C., Wang, H., Han, G., & Zhang, X. (2020). Droidlens: Robust and Fine-Grained Detection for Android Code Smells. In Proceedings - 2020 International Symposium on Theoretical Aspects of Software Engineering, TASE 2020, (pp. 161-168).

Mõškovski, S. (n.d.). Building a tool for detecting code smells in Android application code.

Nguyen, M. D., Huynh, T. Q., & Nguyen, T. H. (2016). Improve the performance of mobile applications based on code optimization techniques using PMD and android lint. In Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 9978 LNAI, (pp. 343-356).

Nguyen, M. D., Huynh, T. Q., & Nguyen, T. H. (2016). Improve the performance of mobile applications based on code optimization techniques using PMD and android lint. In Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 9978 LNAI, (pp. 343-356).

Nguyen, M. D., Huynh, T. Q., & Nguyen, T. H. (2016). Improve the performance of mobile applications based on code optimization techniques using PMD and android lint. In Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 9978 LNAI, (pp. 343-356).

Ogenrwot, D., Nakatumba-Nabende, J., & van Chaudron, M. R. (2020). Comparison of Occurrence of Design Smells in Desktop and Mobile Applications. Retrieved from http://web.engr.oregonstate.edu/Francisco, Calif.). (2013). 1st International Workshop on the Engineering of Mobile-Enabled Systems (MOBS) : Proceedings : May 25, 2013, San Francisco, CA, USA. IEEE.

Ogenrwot, D., Nakatumba-Nabende, J., & van Chaudron, M. R. (2020). Comparison of Occurrence of Design Smells in Desktop and Mobile Applications.

Ournani, Z., Rouvoy, R., Rust, P., & Penhoat, J. (2021). Tales from the Code #1: The Effective Impact of Code Refactorings on Software Energy Consumption.

Ournani, Z., Rouvoy, R., Rust, P., & Penhoat, J. (2021). Tales from the Code #1: The Effective Impact of Code Refactorings on Software Energy Consumption.

Palomba, F., di Nucci, D., Panichella, A., Zaidman, A., & de Lucia, A. (2017). Lightweight detection of Android-specific code smells: The aDoctor project. In SANER 2017 - 24th IEEE International Conference on Software Analysis, Evolution, and Reengineering, (pp. 487-491).

Palomba, F., di Nucci, D., Panichella, A., Zaidman, A., & de Lucia, A. (2019). On the impact of code smells on the energy consumption of mobile applications. *Information and Software Technology*, 105, 43-55.

Palomba, F., di Nucci, D., Panichella, A., Zaidman, A., & de Lucia, A. (2019). On the impact of code smells on the energy consumption of mobile applications. *Information and Software Technology*, 105, 43-55.

Pldi 12 Proceedings Committee. (2013). PLDI 12 Proceedings of the 2012 ACM SIGPLAN Conference on Programming Language Design and Implementation. Association for Computing Machinery.

Rahkema, K., & Pfahl, D. (2020). Comparison of Code Smells in iOS and Android Applications.

Rasool, G., & Ali, A. (2020). Recovering Android Bad Smells from Android Applications. *Arabian Journal for Science and Engineering*, 45(4), 3289-3315.

Rodriguez, A., Mateos, C., & Zunino, A. (2017). Improving scientific application execution on android mobile devices via code refactorings. *Software - Practice and Experience*, 47(5), 763-796.

Skretting, A., & Gronli, T. M. (2020). Baseline for Performance Prediction of Android Applications. In Proceedings - 2020 IEEE International Conference on Big Data, Big Data 2020, (pp. 3304-3310).

Skretting, A., & Gronli, T. M. (2020). Baseline for Performance Prediction of Android Applications. In Proceedings - 2020 IEEE International Conference on Big Data, Big Data 2020, (pp. 3304-3310).

Usman, M., Iqbal, M. Z., & Khan, M. U. (2020). An automated model-based approach for unit-level performance test generation of mobile applications. *Journal of Software: Evolution and Process*, 32(1).

Zhang, P., & Elbaum, S. (2012). Amplifying tests to validate exception handling code. In Proceedings - International Conference on Software Engineering, (pp. 595–605).