An Automated Approach for Detection and Code Refactoring of Mobile Applications to Enhance Performance

Hina Shoaib

September 3, 2022

An Automated Approach for Detection and code Refactoring of Mobile Applications to Enhance Performance

Hina Shoaib
Department of Computing
National university of computer and emerging sciences
I202033@nu.edu.pk

Abstract:
Mobile applications are highly dependent on performance. Performance has become an important aspect on which the quality of applications relies. Detection of problematic code is a way to remove the code smells which automatically improves the performance of the application. If the source code contains bad smells and anti-patterns the performance of the application is compromised. Code smells can directly impact memory, power consumption, and CPU usage. It is identified that the existing literature does not detect 2-3 code smells like “String Concatenation” and Static Views” in android applications. There is also a need to investigate the effect of code smell on performance in mobile applications. Moreover, there is a need to verify empirically that detection has benefits in improving the performance of mobile applications. In this study, we propose an automated approach for the Detection of Code smells in Mobile Applications to enhance Performance. The proposed approach ensures to provide detected code smell with an instance where the smell is detected. Our approach detected the code smell “string concatenation” from the android applications and 6 other smells. Experiment conducted to show the validity of the approach and the impact of the code smell used. The result of the experiment shows a clear difference in improved processing time without using string concatenation. We evaluated results on open-source applications to detect and refactor the smell and the results show the smell exists in the application. It indicates the instances where the smell was detected.

Keywords: Android Code Smell, Detection Tool, Refactoring, Static Code Analysis, Performance.

I. INTRODUCTION

Mobile applications have been increasingly developed and used by society for many years. Android and Apple iOS are two dominating markets for mobile app development [2]. Due to the rapid change in requirements and market demand, developers only work on the functional requirements without using the architecture or model. This causes problems latterly and the quality of the application is compromised [1]. Testing of mobile applications is an important phase of the application development cycle. The basic purpose of software testing is to judge software quality in terms of user acceptance. Testing is involved throughout the development process of software. Software testing has been divided into functional testing, security testing, performance testing, and many others as well. Performance testing provides a detailed analysis of system performance, identification of bottlenecks, and the load or stress range of the system. Most of the time, performance testing is kept at the end of the development process, which causes problems afterward. This causes an impact on the performance of the application [3]. Performance testing is fulfilled by comparing integrated code with non-functional aspects of applications. There is a huge difference in the performance testing of mobile applications and other systems. due to the limitations in resources like memory, energy consumption, UI, and processing power. Performance testing has become a common factor on which application quality is based [4], [5]. Mobile applications run on mobile devices with limited battery lifetime, developers cannot avoid these characteristics. These factors affect the performance of an application. To improve performance and deal with energy consumption, memory, processing time, and battery lifetime there are several ways to develop an application. like model-based implementation, use proper code patterns and styles. Improper way of developing the application cause problem in performance through coding style or pattern as well as UML impact the design and become the reason for design smells. Similarly, the code can also become the reason for the ruin of performance and response time. Detection of that problematic part of code is a way to remove the smelly code which automatically improves the performance of the application [6]. Most studies provide detection details of object-oriented smells that exist in mobile applications. Detection of Android-specific smells is limited in the literature. Few papers provide detection techniques for specific smells from mobile software applications [5]–[13]. These papers provide detection on static analysis of code. Mobile application development is still a fresh and rapidly developing field. In the case of smell detection, the studies contain limited techniques. Researchers are still finding code smells related to mobile applications. Several empirical studies provide detailed knowledge about the impact of bad code smells and different antipatterns which impact mobile applications' non-functional aspects. These smells and antipatterns can impact performance, usability, and maintainability, increasing the complexity and quality of software. Previous literature contains studies named energy consumption or energy leaks, memory utilization, or memory leaks these all come under the umbrella of performance. It reflects the importance of performance and the role of code smells on performance[7]. It motivates the detection of code smells to improve performance. Several approaches exist in the literature, but those are focused on performance testing of GUI and improvement of the design model. GUI provides usability testing and performance of system response time. There is a need to target source code to improve the performance of an application. The application code uses different patterns and coding styles these could result in performance degradation of the application. Mobile applications provide an extensive set of patterns and coding styles. However, the insufficient testing of source code can result in total application performance failure as well as the development effort goes to waste [8]. The need is to provide a mechanism to test and provide evidence whether the application's source code is fulfilling the performance expectation of the end-user, system constraints as well as behave reasonably in response to different user-level and system-level events. The suitability and effectiveness of any approach can be best tested if it applies to real industry projects. The approaches mentioned in the literature for detection and refactoring of code work the based on static code analysis. Several smells need to be detected automatically but the existing tools have limitations in finding few smells. some tools are available and some of them are not openly accessible. A systematic study [4]provides detail about different detection and refactoring tools. According to this study, the tools aDoctor and Paprika can detect some code smells but 2-3 smells are still undetected for example "string concatenation" and "static view". This leads to the gap in the detection that code smells to improve the performance of the application. Through a literature review, it is identified that the following are some gaps related to the detection of mobile applications code smells. Firstly, there is no comprehensive taxonomy of mobile applications' bad smells. There is also a need to investigate the effect of undetected code smells on performance in mobile applications.

In this paper, we proposed an automatic approach for the detection of code smell to ensure better performance of mobile applications. The proposed approach ensures to provide a detection and refactoring mechanism. we had to identify the list of bad code smells that impact the performance of mobile applications from literature and catalog them into a taxonomy. Secondly, we identified undetected smells that impact energy consumption, processing time, and memory consumption. Our approach detects the code smell and provides the name of the instance where the smell is found. We can apply it to various mobile applications to evaluate the significance.

This paper contains following contributions

- To define the taxonomy of performance-based mobile applications code smells and detection rules
- To identify the appropriate rule for detection of "String Concatenation".
- To detect "string Concatenation" using an automated detection approach.
- To evaluate the impact of a detected smell on performance and evaluation of the proposed approach by applying it to different open-source mobile applications.

II. BACKGROUND AND MOTIVATION

Code smells are not errors in the code. a code smell is a violation of the fundamentals of programming. Code smells are design or architecture flaws and bad programming practices. It is not an error directly but it indicates a deeper problem in the code[9]. It could be in the form of dead code, unnecessary use of code, empty method, If statements, etc. All these are examples of code smells that impact the application badly in terms of memory, energy, or time. Developing an application or software does not mean coding. It includes the optimized way of developing an application that is performance efficient. Even though, if the software is having smells that do not mean it will not work. It will do all the tasks and provide an output. But the issue that can arise is the process could be slow and the quality of the application could be compromised. Different types of smells exist in the code for example design smell, code smell, network smell, and database smells. that Mobile applications also have different smells than web applications. Some of them are common and many are different. Strings are the common way of handling text and data in a program. When it is in terms of java development the data type which is most commonly used for reading data or saving it is a string. Strings are also used to create big data but, in that case, string concatenation can be used. Davide and Rick [45] conducted a detailed study to identify "worst smells". The main contribution to the paper is to provide a detailed list of worst and non-worst smells. They surveyed 71 expert developers and five telephonic interviews. They collected 314 smells from 27 large Apache open-source projects. This study provides 314 code smells in a catalog and 80 are learned to be the worst. This study also claims that the frequency of occurrence and change proneness is different in worst and non-worst smells. The reason behind conducting the study is to list down the worst smells to help the developer to improve the quality of applications by removing these smells.

Rodriguez *et al. [*12] studied that increasing the use of mobile also increases the demand for mobile applications. Due to fewer resources, mobile devices have to suffer in many ways. Similarly, if the application is not developed by keeping programming fundamentals in mind this will leads to intensive problems. There is a certain list of programming primitives found that help in developing a quality application. This study discusses on important programming practice "string handling" and its impact on performance. The study shows that using string concatenation vs string builder shows a significant difference in the performance.

Dong kwan kim [4] stated in this research that coding style and selection or patterns plays important role in enhancing the performance. This study provides best practices for Android development. To check the impact of these practices' evolution done on different applications and how much CPU time is consumed. This study also provides the use of string Buffer as a best practice. Nguyen *et al.* [12] investigated the rules and analysis process of java source code using PMD and Android Lint. This study provides an automated analyzer and code refactoring tool which works on a rule-based approach. This study listed 49 different rules using these rules in programming will help in decreasing battery consumption. Seven rules are implemented in this paper on two different applications and that shows optimal results in terms of performance. This study also mentioned "string concatenation" as bad programming practice. Yang *et al*. [17], have proposed an approach to discover and compute common causes of the poor response time of Mobile applications. They extend the delay for problematic operations, by using the test magnification approach, to establish the effects of exclusive actions that can be observed by end-users. Performance testing of mobile applications contains several challenges like different operating systems and different mobile devices impact the performance of an application. secondly, the generation of test cases to achieve the coverage is challenging [18]. Usman *et al*[1] . have proposed a product-line-based approach that will help to deal with different versions of the system and also provide a model-based approach to improve the performance. It provides an automated approach for unit-level test case generation in the mobile application. By performing it on two different applications. It captures the energy consumption, memory allocation, processing time, and so on. The result shows the improvement in the performance of the application before and after applying the UML-based model-based approach. Mobile application performance testing is a dynamic research area to detect code-related issues of performance. To auto-generate the test cases for performance evaluation and fault detection in software. There are some plug-ins of the eclipse available that automatically generate the test case. Test-Driven development helps to find the fault in software at the unit level [19]. Performance testing of mobile applications can be improved by the detection of bad code or code smells that occur in source code. Many pieces of research investigated the type of bad code smells. They provide detail of different tools that help in finding the code smells that impact the performance [20]. The existing paper contains detail about different tools and techniques that help to optimize the code through a variety of APIs plug-ins or detection tools like LoadRunner or Robotium [21], [[22]]. Chouchane *et al.*[27] proposed an approach to detect the presentation layer code smells. How do these smells impact the performance of the application? This study uses two tools to detect aesthetic defects and code smells. One tool is PLAIN which is for aesthetic defects and the other is an Android UI detector. The evaluation is done on 120 android applications with 8480 GUI's. It investigates performance impact empirically on a different machine learning algorithm. It proposes a prediction model for the detection of smells from the presentation layer. 15 android smells are detected which impacts the presentation layer. The results reflect that the code smells and aesthetic defects impact the performance of android applications.

A large number of studies have been conducted for code smell identification and detection. Amalfitano *et al.* [11] have proposed an automated technique for the detection of memory leaks. This study focused on the FunesDroid tool. It tests every activity lifecycle to explore the possible leaks by comparing before and after event execution states. This study is exploratory and works on the phenomena of black-box testing. Memory leaks occur due to bad programming practices. This can cause temporary or permanent memory leak issues.

Bhargav and Carlo[10] proposed an automated approach for the detection and fixing of resource leaks in android applications. PLUMBDROID tool is used for fixing and repair of resource leaks. The tool is based on static analysis or source code. This is evaluated from the base of nine android applications. This study is based on experimental evaluation through which it provides the control flow to detect the resource leaks. The limitation of this study experiment is it generates false-positive results. This study evaluates the tool's performance in repairing resource leaks and fixing performance bugs. Palomba *et al.* [6]presented an automated aDoctor tool for the detection of android code smells. This tool detects 15 android specific code smells. This study also includes an empirical investigation to validate the tool. The tool is evaluated on 18 different android source codes. It concluded that the tool detects design flaws from code. Empirical evaluation highlights that detection of two smells *Data Transmission Without Compression* and *Inefficient SQL Query* lower through this tool. Dong kwan kim [4] stated in this research that coding style and selection or patterns plays important role in enhancing the performance. This study provides best practices for Android development. To check the impact of these practices' evolution done on different applications and how much CPU time is consumed. This study also provides the use of string Buffer as a best practice. Nguyen et al. [9] investigated the rules and analysis process of java source code using PMD and Android Lint. This study provides an automated analyzer and code refactoring tool which works on a rule-based approach. This study listed 49 different rules using these rules in programming will help in decreasing battery consumption. Seven rules are implemented in this paper on two different applications and that shows optimal results in terms of performance. This study also mentioned "string concatenation" as bad programming practice.

III. METHODOLOGY

This section presents the detail of an automated approach for the detection of android applications code smell. The overview of the proposed approach. The steps with detailed descriptions and figures to create a clear and concise understanding. It also contains the detail of the proposed tool architecture and working details. The approach takes android mobile applications as input. The output of the approach is a detected smell with the name of the class where the smell was found and the total number of instances where the smell was detected.

To tackle the issue of large application smell detection, we build a plugin that is integrated with android studio and helps in the detection of code smells from android applications. For this process, we have to analyze the application programmatically. Static code analysis will help in this regard. In this plugin, we can write customize rules for each smell. To create a plugin, we use guidelines from the literature. aDoctor plugin provides the base for our approach. The plugin is integrated with android studio. The development of the plugin is core on IntelliJ IDE. It uses the java libraries, API of java language, and abstract syntax tree for analysis of code. It also has a proposal and analyzer which defines the customized rules.

The proposed approach starts with the detection of code smell in an android application, which is mentioned in the figure. A rule is set for the detection of that smell in the proposed plugin. For the detection of that smell, the plugin consists of the following steps, which work as the main components in detection. 1) Examination of Android applications 2) Recognize a rule violation in an application; 3) provide smell details such as the number of instances and the name of an instance where the smell is located.

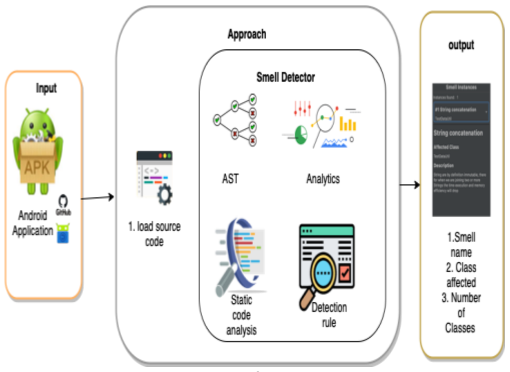


Figure: Overview of proposed approach

Static analysis of code is used for the detection of code smells. The rule is written for the detection of smell through an android's source code. The approach is written in Java and only Java-specific applications are used for the detection of smells. These smells badly impact the performance of a mobile application in terms of memory, energy, and processing time. This approach contributes to the detection of the new smell. The approach is based on a plugin of the Android studio which can automatically detect the smell from the classes of Android source code.

a) Approach Component Detail:

The approach consists of different steps for the detection of smells, and the internal working mechanism for the approach is as follows: The source

1. code consists of Java and XML files. It first went to AST for analysis.
2. The AST process categorizes the code elements into different types like class instance creation, method declaration, method invocation, and variable declaration.
3. After AST completion, the smell rule is applied to it. When an element fails to meet the rule's criteria, That is considered smelling.
4. Detection identifies the smell and saves the class details like name and a total number of classes where the smell is detected and delivered to the user.

IV. IMPLEMENTATION

Tool's Architecture

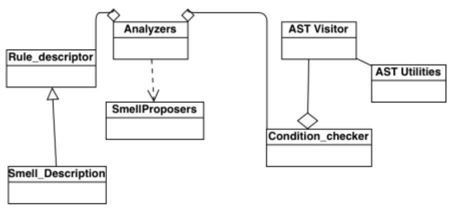
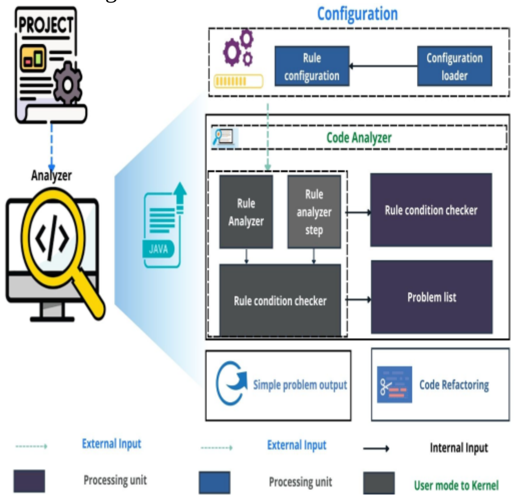


Figure: Class diagram

The plugin is implemented with IntelliJ SDK. It provides a set of libraries that allows extension by creating a customized plugin, language, or IDE. This plugin is for android studio. The proposed plugin is based on two main layers 1) presentation layer and 2) Application layer/ business logic.



The presentation lay consist of basic controls. A GUI with a dialog box that shows detected smells and their details. For that dialog system, java swing is used. The core logic for the dialog is implemented in CoreDriver class. It contains the logic for different controls of dialogs like Start Dialog, NoSmell, Abort, and Smell list.

The business logic is implemented in the Application layer. It contains Analyzer, Analytics, and AST. i) Analyzer performs smell detection, ii) Analytics provide statistical analysis of the like type of smells user selected from the checklist menu. This can further be used for the investigation. and iii) AST checks the elements of source code and then applies the rule written for the detection of elements.

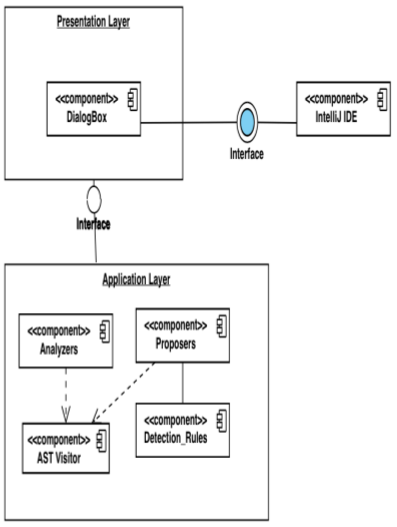


Figure: Architecture Diagram

V. EVALUATION:

In this section, experiments and results are described. It also contains a detailed evaluation of the proposed approach by answering the research questions in detail. The evaluation of the proposed approach is done by doing the following steps: 1) Conducting an Experiment to evaluate the effectiveness of the detected smell, 2) applying the detection rule to the different applications, and 3) Detected results of the smell "string concatenation".An experiment was conducted to check the code smell effectiveness. The reason to conduct this experiment is to check whether the detected smell has any kind of impact on performance. For this purpose, the environment needed to build an android application which will be tested later on is as follows: Android studio Bumblebee | 2021.1.1 patch 3, Android Gradle plugin version 7.1.3, Gradle version 7.2, Android Studio default JDK version 11.0.11, and Android Virtual Device (AVD) Nexus 5X API 27 (Portrait). The detailed settings of the tested application are defined in below Table 7. The application used for the detailed experiment is "TestAndroid".

Software version	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

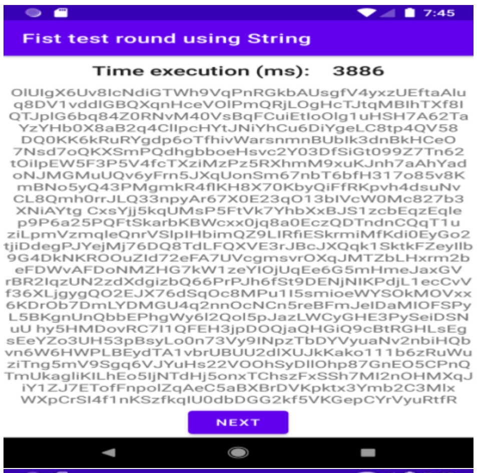
The application is built to check the following features:

1. To check the performance of the android application while using string vs string builder.
2. To check that our proposed approach can detect the smell correctly or not
3. How much difference will be occurred when we use string vs string builder.

We develop an application in android studio using above mentioned details. This application is built to test and validate the code smell. This experiment validates that string builder is better in performance than string.

Test case No	**Test Case**	
	Number of strings	**Length of string**
T1	22	2222
T2	222	221
T3	2212	456

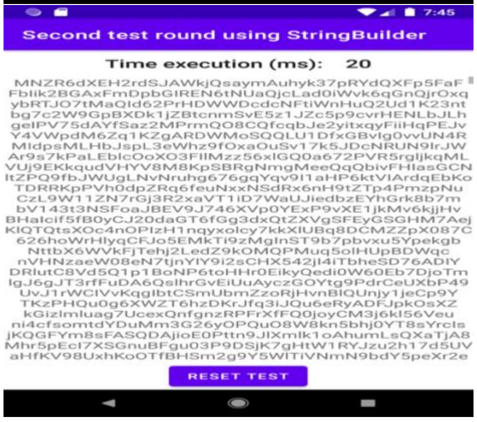
String concatenation uses the "+" operator and the execution time grows four times. The complexity of concatenation is in a loop of the nth iteration would be O(n^2). While if we look at the string builder result it is 20ms which is very less compared to the string. In each string builder *append ()* will take O (1) as constant time. So, the complexity of the process will be calculated O(n).



Fist test round using String

Time execution (ms): 3886

```
OlUIgX6Uv8IcNdiGTWh9VqPnRGkbAUsgfV4yxzUEftaAlu
q8DV1vddlGBQXqnHceVOlPmQRjLOgHcTJtqMBIhTXf8I
QTJplG6bq84Z0RNvM40VsBqFCuiEtIoOlg1uHSH7A62Ta
YzYHb0X8aB2q4CllpcHYtJNiYhCu6DiYgeLC8tp4QV58
DQ0KK6kRuRYgdp6oTfhivWarsnmnBUbIk3dnBkHCeO
7Nsd7oQKXSmPQdhgbboeHsvc2Y03DfSiGt099Z7Tn62
tOilpEW5F3P5V4fcTXziMzPz5RXhmM9xuKJnh7aAhYad
oNJMGMuUQv6yFrn5JXqUonSm67nbT6bfH317o85v8K
mBNo5yQ43PMgmkR4flKH8X70KbyQiFfRKpvh4dsuNv
CL8Qmh0rrJLQ33npyAr67X0E23qO13bIVcW0Mc827b3
XNiAYtg CxsYjj5kqUMsP5FtVk7YhbXxBJS1zcbEqzEqle
p9P6a25PQFtSkarbKBWcxx0jq8a0EczQDTndnCQqT1u
ziLpmVzmqleQnrVSlpIHbimQZ9LIRfiESkrmiMfKdi0EyGo2
tjiDdegPJYejMj76DQ8TdLFQXVE3rJBcJXQqk1SktkFZeyIlb
9G4DkNKROOuZId72eFA7UVcgmsvrOXqJMTZbLHxrm2b
eFDWvAFDoNMZHG7kW1zeYIOjUqEe6G5mHmeJaxGV
rBR2IqzUN2zdXdgizbQ66PrPJh6fSt9DENjNIKPdjL1ecCvV
f36XLjgygQO2EJX76dSqOc8MPu1I5smioeWYSOkMOVxx
6KDrOb7DmLYDMGU4q2nnOcNCn5reBFmJelDaMIOFSPy
L5BKgnUnQbbEPhgWy6l2Qol5pJazLWCyGHE3PySeiDSN
uU hy5HMDovRC7I1QFEH3jpDOQjaQHGiQ9cBtRGHLsEg
sEeYZo3UH53pBsyLo0n73Vy9INpzTbDYVyuaNv2nbiHQb
vn6W6HWPLBEydTA1vbrUBUU2dlXUJkKako111b6zRuWu
ziTng5mV9Sgq6VJYuHs22VOOhSyDllOhp87GnEO5CPnQ
TmUkagliKILhEo5IjNTdHj5onxTChszFxSSh7MI2nOHMXqJ
iY1ZJ7ETofFnpolZqAeC5aBXBrDVKpktx3Ymb2C3Mlx
WXpCrSI4f1nKSzfkqIU0dbDGG2kf5VKGepCYrVyuRtfR
```



Second test round using StringBuilder

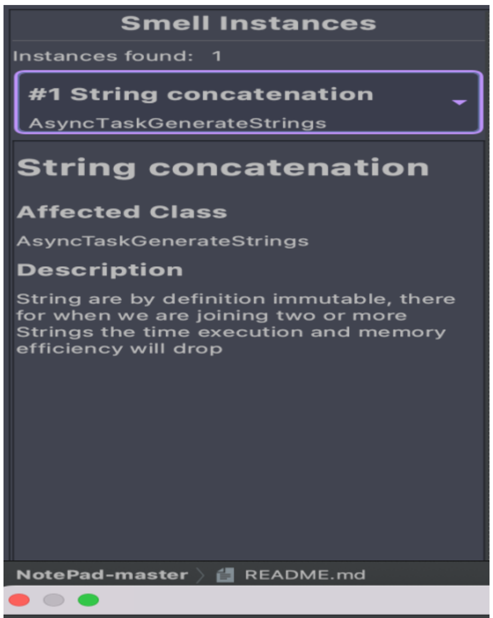
Time execution (ms): 20

```
MNZR6dXEH2rdSJAWkjQsaymAuhyk37pRYdQXFp5FaF
Fblik2BGAxFmDpbGIREN6tNUaQjcLad0iWvk6qGnQjrOxq
ybRTJO7tMaQld62PrHDWWDcdcNFtiWnHuQ2Ud1K23nt
bg7c2W9GpBXDk1jZBtcnmSvE5z1JZc5p9cvrHENLbJLh
gelPV75dAYfSaz2MPrmQO8CQfcqbJe2yitxqyFiiHqPEJv
Y4VWpdM6Zq1KZgARDWMoSQQLU1DfxGBvlg0vvUN4R
MIdpsMLHbJspL3eWhz9fOxaOuSv17k5JDcNRUN9lrJW
Ar9s7kPaLEblcOoXO3FIIMzz56xIGQ0a672PVR5rgljkqML
VUj9EKkqudVHYV8M8KpSBRgNmgMeeQqQbivFHIasGCN
ItZPQ9fbJWUgLNvNruhg676gqYqv9I1aHP6ktVIArdqEbKo
TDRRKpPVh0dpZRq6feuNxxNSdRx6nH9tZTp4PmzpNu
CzL9W11ZN7rGj3R2xaVT1iD7WaUJiedbzEYhGrk8b7m
bV143t3NSFoaJBEV9J746XVp0YExP9vXE1jkMv6kjjHv
BHalcif5fB0yCJ20daGT6fGg3dxQtZXVgSFEyGSGHM7Aej
KIQTQtsXOc4nOPIzH1nqyxolcy7kkXIUBq8DCMZZpX087C
626hoWrHlyqCFJo5EMkTi9zMgInST9b7pbvxu5Ypekgb
NttbX6WVkFjTehj2LedZ9kOMQPMuq5olHUpBDWqc
nVHNzaeW08eN7tjnYIY9i2sCHX542jI4iTbheSD76ADIY
DRlutC8Vd5Q1p1BoNP6toHHr0EikyQedi0W60Eb7DjoTm
IgJ6gJT3rfFuDA6QslhrGvEiUuAyczGOYtg9PdrCeUXbP49
UvJ1rWCIVvKqglbtCSmUbmZzoRjHvnBlQUnjy1jeCp9Y
TKzPHQu0g6XWZT6hzDKrJfq3iJQu6eRyADFJpkOsXZ
kGizlmluag7UcexQnfgnzRPFrXfFQ0joyCM3j6kl56Veu
ni4cfsomtdYDuMm3G26yOPQuO8W8kn5bhj0YT8sYrcls
jKQGFYm8sFASQDAjioE0Pttn9JlXmlk1oAhumLsQXaTjA8
Mhr5pEcl7XSGnuBFgu03P9DSjK7gHtW1RYJzu2h17d5UV
aHfKV98UxhKoOTfBHSm2g9Y5WlTiVNmN9bdY5peXr2e
```

After Validating that the string builder is more efficient than the string or string concatenation. We applied the detection rule to open-source applications. Our plugin can detect 7 smells but we are evaluating the plugin only on string concatenation. After importing the android application source code into the android studio. we have to select the refactoring tab from the Refactor tab of android studio. It will open up with the window. Select the checkbox from the dialog box and click the run button. It will start analyzing the source code and detect the corresponding smell if exist in the code.

Application	#Commits	Languages
Apg	4376	Java
Notepad	1522	Java, python
Al muazzin	133	Java
Bitcoin-wallet	4142	Java
A photo Manager	1126	Java, batchfile

The detected smell "string concatenation" is from the application TestAndroid. The Smell detected the class "AsyncTaskGenerationStrings".
It also shows that there is only one instance in which this smell exists.
NotePad-master application is an open-source application. It is also taken from GitHub. The smell of "string concatenation" is detected in the class "OrgConverter".



Smell Instances

Instances found: 1

#1 String concatenation
AsyncTaskGenerateStrings

String concatenation

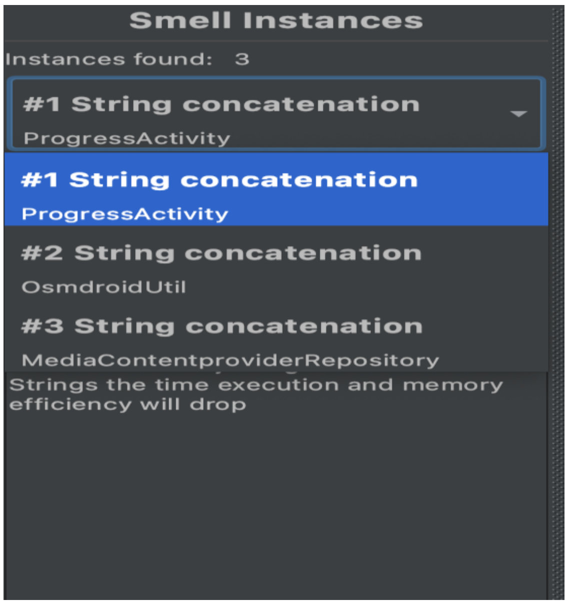
Affected Class

AsyncTaskGenerateStrings

Description

String are by definition immutable, there for when we are joining two or more Strings the time execution and memory efficiency will drop

NotePad-master 〉 README.md



Smell Instances

Instances found: 3

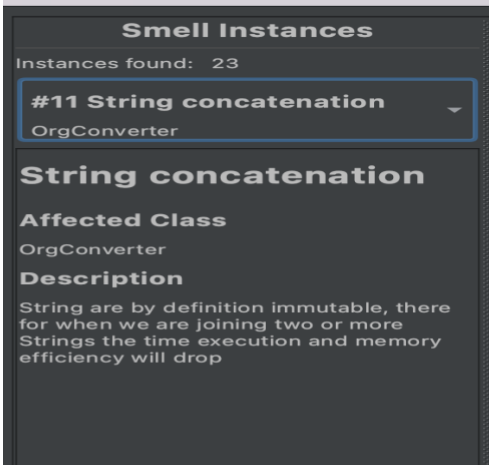
#1 String concatenation
ProgressActivity

#1 String concatenation
ProgressActivity

#2 String concatenation
OsmdroidUtil

#3 String concatenation
MediaContentproviderRepository

Strings the time execution and memory efficiency will drop



Smell Instances

Instances found: 23

#11 String concatenation
OrgConverter

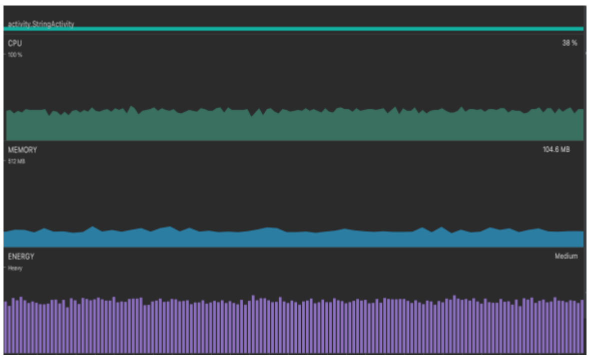
String concatenation

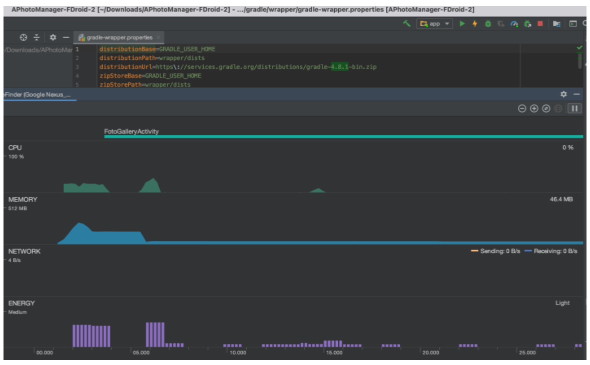
Affected Class

OrgConverter

Description

String are by definition immutable, there for when we are joining two or more Strings the time execution and memory efficiency will drop

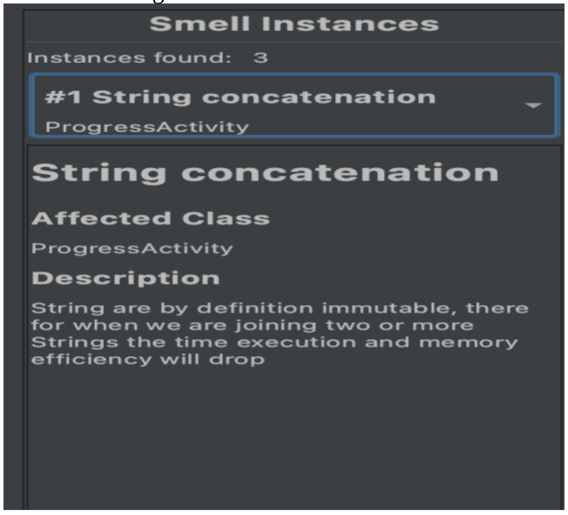




APhotoManager-FDroid-2 [~/Downloads/APhotoManager-FDroid-2] - .../gradle/wrapper/gradle-wrapper.properties [APhotoManager-FDroid-2]

```
distributionBase=GRADLE_USER_HOME
distributionPath=wrapper/dists
distributionUrl=https\://services.gradle.org/distributions/gradle-4.8.1-bin.zip
zipStoreBase=GRADLE_USER_HOME
zipStorePath=wrapper/dists
```

A photo Manager application is an open-source application. It is taken from GitHub also available on F-Droid . The smell of “string concatenation” is detected in the classes “ProgressActivity” , MediaContentproviderRepository” and OsmdroidUtil shown in the Figure.



Smell Instances

Instances found: 3

#1 String concatenation
ProgressActivity

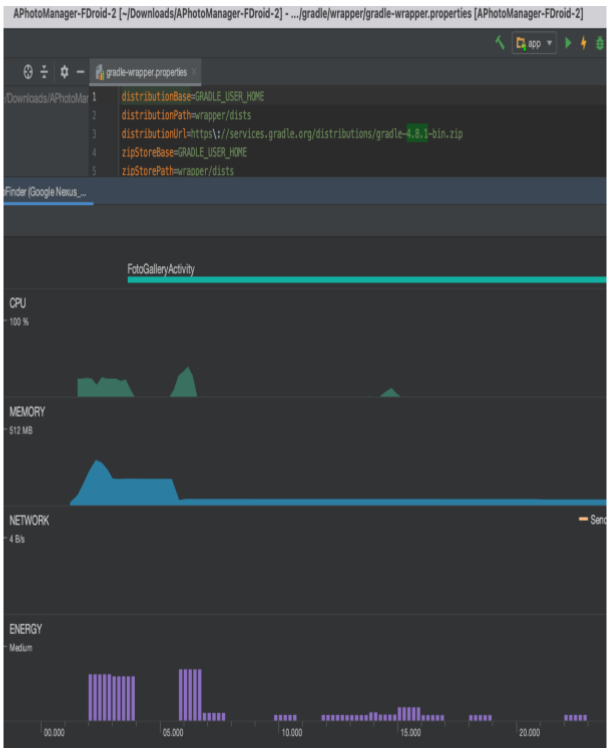
String concatenation

Affected Class

ProgressActivity

Description

String are by definition immutable, there for when we are joining two or more Strings the time execution and memory efficiency will drop


APhotoManager-FDroid-2 [~/Downloads/APhotoManager-FDroid-2] - .../gradle/wrapper/gradle-wrapper.properties [APhotoManager-FDroid-2]

```
distributionBase=GRADLE_USER_HOME
distributionPath=wrapper/dists
distributionUrl=https\://services.gradle.org/distributions/gradle-4.8.1-bin.zip
zipStoreBase=GRADLE_USER_HOME
zipStorePath=wrapper/dists
```

VI Conclusion

Mobile phones are an important part of life nowadays. It replaces computers in light and daily work & entertainment activities. The daily increase in the use of mobile also increases the demand for applications to use. This needs rapid development of applications to compete with the market. At the same time, users do not want to compromise on quality. Quality is the main aspect when it comes to the long-term market capturing strategy. In the development process, most of the time developer neglects the fundamentals of programming. This leads to the performance degradation issue. As code smells will introduce, code smells are bad programming practices. This is a need for a solution to provide quality and performance upgrading tools. Detection of problematic code is a way to remove the code smells which automatically improves the performance of the application. If the source code contains bad smells and anti-patterns the performance of the application is compromised. Code smells can directly impact memory, power consumption, and CPU usage. It is identified that the existing literature does not detect 2-3 code smells like "String Concatenation" and Static Views" in android applications. There is also a need to investigate the effect of code smell on performance in mobile applications. Moreover, there is a need to verify empirically that detection has benefits in improving the performance of mobile applications. In this study, we propose an automated approach for the Detection of Code smells in Mobile Applications to enhance Performance. The proposed approach ensures to provide detected code smell with an instance where the smell is detected. Our approach detected the code smell "string concatenation" from android applications. We experimented to show the validity of the approach and the impact of code smell used. The results of the experiment show a clear difference in improved processing time without using string concatenation. We also use 3 open-source applications to detect the smell and the results show the smell exists in the application. It indicates the instances where the smell was detected.

Limitations And Future Work: This study has limitations that our thesis doesn't provide detection and refactoring for all of smells on same platform. It is also limited to provide multiple class refactoring of smell. And whose refactoring might ask to modify more than one class. In that case, other than the internal core logic, the GUI of tool should be updated as well, to properly handle these cases. In the future, We are aiming to use other types of smells like network, UI and database, etc. to provide a better perspective on software quality development.

VI. References

[1] M. Usman, M. Z. Iqbal, and M. U. Khan, "An automated model-based approach for unit-level performance test generation of mobile applications," *Journal of Software: Evolution and Process*, vol. 32, no. 1, Jan. 2020, doi: 10.1002/smr.2215.

[2] G. Rasool and A. Ali, "Recovering Android Bad Smells from Android Applications," *Arabian Journal for Science and Engineering*, vol. 45, no. 4, pp. 3289–3315, Apr. 2020, doi: 10.1007/s13369-020-04365-1.

[3] M. D. Nguyen, T. Q. Huynh, and T. H. Nguyen, "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)*, 2016, vol. 9978 LNAI, pp. 343–356. doi: 10.1007/978-3-319-49046-5_29.

[4] I. Fatima, H. Anwar, D. Pfahl, and U. Qamar, "Tool support for green android development: A systematic mapping study," in *ICSOFT 2020 - Proceedings of the 15th International Conference on Software Technologies*, 2020, pp. 409–417. doi: 10.5220/0009770304090417.

[5] C. Mao, H. Wang, G. Han, and X. Zhang, "Droidlens: Robust and Fine-Grained Detection for Android Code Smells," in *Proceedings - 2020 International Symposium on Theoretical Aspects of Software Engineering, TASE 2020*, Dec. 2020, pp. 161–168. doi: 10.1109/TASE49443.2020.00030.

[6] F. Palomba, D. di Nucci, A. Panichella, A. Zaidman, and A. de Lucia, "Lightweight detection of Androidspecific code smells:

The aDoctor project," in *SANER 2017 - 24th IEEE International Conference on Software Analysis, Evolution, and Reengineering*, Mar. 2017, pp. 487–491. doi: 10.1109/SANER.2017.7884659.

[7] S. Mõškovski, "Building a tool for detecting code smells in Android application code."

[8] S. Habchi, N. Moha, and R. Rouvoy, "Android Code Smells: From Introduction to Refactoring," 2020, [Online]. Available: http://arxiv.org/abs/2010.07121

[9] Pldi 12 Proceedings Committee, *Pldi 12 Proceedings of the 2012 Acm Sigplan Conference on Programming Language Design and Implementation.* Association for Computing Machinery, 2013.

[10] D. Amalfitano, V. Riccio, P. Tramontana, and A. R. Fasolino, "Do Memories Haunt You? An Automated Black Box Testing Approach for Detecting Memory Leaks in Android Apps," *IEEE Access*, vol. 8, pp. 12217–12231, 2020, doi: 10.1109/ACCESS.2020.2966522.

[11] B. N. Bhatt and C. A. Furia, "Automated Repair of Resource Leaks in Android Applications," Mar. 2020, [Online]. Available: http://arxiv.org/abs/2003.03201

[12] M. U. Khan, S. U. J. Lee, S. Abbas, A. Abbas, and A. K. Bashir, "Detecting Wake Lock Leaks in Android Apps Using Machine Learning," *IEEE Access*, vol. 9, pp. 125753–125767, 2021, doi: 10.1109/ACCESS.2021.3110244.

[13] S. Boutaib, S. Bechikh, F. Palomba, M. Elarbi, M. Makhlouf, and L. ben Said, "Code smell detection and identification in imbalanced environments," *Expert Systems with Applications*, vol. 166, Mar. 2021, doi: 10.1016/j.eswa.2020.114076.

[14] P. Zhang and S. Elbaum, "Amplifying tests to validate exception handling code," in *Proceedings - International Conference on Software Engineering*, 2012, pp. 595–605. doi: 10.1109/ICSE.2012.6227157.

[15] A. Degu, "Android Application Memory and Energy Performance: Systematic Literature Review," vol. 21, no. 3, pp. 20–32, doi: 10.9790/0661-2103052032.

[16] M. Hort, M. Kechagia, F. Sarro, and M. Harman, "A Survey of Performance Optimization for Mobile Applications," *IEEE Transactions on Software Engineering*, 2021, doi: 10.1109/TSE.2021.3071193.

[17] Institute of Electrical and Electronics Engineers. and Calif.) International Conference on Software Engineering (35th : 2013 : San Francisco, *2013 1st International Workshop on the Engineering of MobileEnabled Systems (MOBS) : proceedings : May 25, 2013 San Francisco, CA, USA.* IEEE, 2013.

[18] P. Kong, L. Li, J. Gao, K. Liu, T. F. Bissyandé, and J. Klein, "Automated testing of Android apps: A systematic literature review," *IEEE Transactions on Reliability*, vol. 68, no. 1, pp. 45–66, 2019, doi: 10.1109/TR.2018.2865733.

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

[20] A. Rodriguez, C. Mateos, and A. Zunino, "Improving scientific application execution on android mobile devices via code refactorings," *Software - Practice and Experience*, vol. 47, no. 5, pp. 763–796, May 2017, doi: 10.1002/spe.2419.

[21] D. Kwan Kim, "Towards Performance-Enhancing Programming for Android Application Development 39," *International Journal of Contents*, vol. 13, no. 4, 2017, doi: 10.5392/IJoC.2017.13.4.039.

[22] M. D. Nguyen, T. Q. Huynh, and T. H. Nguyen, "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)*, 2016, vol. 9978 LNAI, pp. 343–356. doi: 10.1007/978-3-319-49046-5_29. 2020, pp. 3304–3310. doi: 10.1109/BigData50022.2020.9377882.

[23] T. Das, M. di Penta, and I. Malavolta, "Characterizing the evolution of statically-detectable performance issues of Android apps," *Empirical Software Engineering*, vol. 25, no. 4, pp. 2748–2808, Jul. 2020, doi: 10.1007/s10664-019-09798-3.

[24] S. Boutaib, S. Bechikh, F. Palomba, M. Elarbi, M. Makhlouf, and L. ben Said, "Code smell detection and identification in imbalanced environments," *Expert Systems with Applications*, vol. 166, 2021, doi: 10.1016/j.eswa.2020.114076.

[25] M. Chouchane, M. Soui, and K. Ghedira, "The impact of the code smells of the presentation layer on the diffuseness of aesthetic defects of Android apps," *Automated Software Engineering*, vol. 28, no. 2, Nov. 2021, doi: 10.1007/s10515-021-00297-8.

[26] G. Hecht, O. Benomar, R. Rouvoy, N. Moha, and L. Duchien, "Tracking the

software quality of android applications along their evolution," in *Proceedings - 2015 30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015*, Jan. 2016, pp. 236–247. doi: 10.1109/ASE.2015.46.

[27] F. Palomba, D. di Nucci, A. Panichella, A. Zaidman, and A. de Lucia, "On the impact of code smells on the energy consumption of mobile applications," *Information and Software Technology*, vol. 105, pp. 43–55, Jan. 2019, doi: 10.1016/j.infsof.2018.08.004.

[28] D. Ogenrwot, J. Nakatumba-Nabende, and M. R. v Chaudron, "Comparison of Occurrence of Design Smells in Desktop and Mobile Applications," 2020. [Online]. Available: http://web.engr.oregonstate.edu/

[29] K. Rahkema and D. Pfahl, "Comparison of Code Smells in iOS and Android Applications," 2020. [Online]. Available: http://ceur-ws.org

[30] O. Hamdi, A. Ouni, M. Cinnéide, and M. W. Mkaouer, "A longitudinal study of the impact of refactoring in android applications," *Information and Software Technology*, vol. 140, Dec. 2021, doi: 10.1016/j.infsof.2021.106699.

[31] S. Habchi, N. Moha, and R. Rouvoy, "Android Code Smells: From Introduction to Refactoring," Oct. 2020, [Online]. Available: http://arxiv.org/abs/2010.07121

[32] M. A. Alkandari, A. Kelkawi, and M. O. Elish, "An Empirical Investigation on the Effect of Code Smells on Resource Usage of Android Mobile Applications," *IEEE Access*, vol. 9, pp. 61853–61863, 2021, doi: 10.1109/ACCESS.2021.3075040.

[33] Effective Impact of Code Refactorings on Software Energy Consumption," 2021. [Online]. Available: https://hal.archivesouvertes.fr/hal-03202437

[34] Z. Ournani, R. Rouvoy, P. Rust, and J. Penhoat, "Tales from the Code #1: The Effective Impact of Code Refactorings on Software Energy Consumption," 2021. [Online]. Available: https://hal.archivesouvertes.fr/hal-0320243