Master Thesis

STATISTICAL ANALYSIS OF ANDROID APPS: A STUDY OF LIFECYCLE DEVELOPMENT PATTERNS

Prepared By

Noura Hoshieah

Supervisor

Dr. Samer Zein

This Thesis was submitted in partial fulfillment of the requirements for the Master's Degree in Software Engineering From the Faculty of Information Technology and Engineering

March 2018

Statistical Analysis of Android Apps: A Study of Lifecycle Development Patterns. By Noura Hoshieah

Approved by the thesis committee:

_____

Dr. Samer Zein, Birzeit University

_____

Dr. Majdi Mafarja, Birzeit University

_____

Dr. Nariman Ammar, Birzeit University

_____

Date Approved:

Declaration of Authorship

I, Noura Hoshieah, declare that this thesis titled, "Statistical Analysis of Android Apps: A Study of Lifecycle Development Patterns" and the work presented in it are our own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given and with the exception of such quotations, this thesis is entirely our own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by ourselves jointly with others, I have made clear exactly what was done by others and what I have contributed ourselves.

Signed: Date:

Abstract

Statistical Analysis of Android Apps: A Study of Lifecycle Development Patterns
By Noura Hoshieah

Building robust Android apps is a non-trivial task that requires skilled developers to understand the different peculiarities of such apps. However, among the Android developer community, a large fraction is considered to be novice and inexperienced developers. One of the main peculiarities in Android app development is a lifecycle model. A developer needs to have a deep understanding of the different lifecycle states and callback methods that an Android activity can go through during its runtime. These callback methods are called by the system whenever the activity changes its state. The developer on one hand needs to override these callback methods correctly to avoid app memory leaks and data loss. Statistical analysis of software applications provides actionable insights and helps to understand how applications were really built. Although there have been lots of studies focusing on analyzing of Android apps in the areas of testing, quality, design, privacy, and security; there are no studies focus on lifecycle development practices thus far.

In this thesis, 842 open-sourced Android apps containing 5577 activities were analyzed to explore and understand how Android developers actually comply with best practices regarding the activity lifecycle model. A tool named Statistical Analysis of Android Lifecycle (SAALC) was developed that is capable to analyze Android activities and extracting valuable information about lifecycle callback methods usage. The generated results show, which callback methods are mostly implemented, what they are implementing for, and the nature of code they contain. More specifically, the results show an incorrect implementation of the callback methods and incorrect acquiring and releasing of a system's resources. The findings suggest that a relatively large fraction of Android developers who didn't well understand the lifecycle model. This research also compares the results obtained with best practices and state-of-the-art.

الملخص

تحليل أنماط دورة حياة تطبيقات الأندرويد

إعداد : نورا حوشية

يُعد إنشاء تطبيقات الأندرويد المتينة مهمة غير عادية تتطلب من المطورين المحترفين فهم الخصائص المختلفة لمثل هذه التطبيقات. ومع ذلك ، من بين مجتمع مطوّري برامج الأندرويد ، هنالك جزء كبير من المطورين المبتدئين وغير المحترفين. واحدة من الخصائص الرئيسية في تطوير تطبيق الأندرويد هي نموذج دورة الحياة. يحتاج المطور إلى فهم عميق لحالات دورة الحياة المختلفة وطرق رد الاتصال التي يمكن أن يمر بها نشاط الأندرويد خلال وقت التشغيل. بحيث يتم استدعاء أساليب رد الاتصال هذه بواسطة النظام عندما يغير النشاط حالته. و يحتاج المطور إلى تعزيز أساليب رد الاتصال هذه بشكل صحيح لتجنب تسرب ذاكرة التطبيق وفقدان البيانات. يوفر التحليل الإحصائي لتطبيقات البرامج إحصاءات قابلة للتنفيذ ويساعد على فهم كيفية بناء التطبيقات بالفعل. وعلى الرغم من وجود الكثير من الدراسات التي تركز على تحليل تطبيقات الأندرويد في مجالات الاختبار والجودة والتصميم والخصوصية والأمان ؛ لا توجد دراسات تركز على ممارسات تطوير دورة الحياة حتى الآن.

في هذه الأطروحة ، تم تحليل 842 من تطبيقات الأندرويد المفتوحة المصدر التي تحتوي على 5577 نشاطاً لاستكشاف وفهم كيفية التزام مطوري برامج الأندرويد بأفضل الممارسات المتعلقة بنموذج دورة حياة النشاط. وتم تطوير أداة تسمى التحليل الإحصائي لدورة حياة الأندرويد (سالس) وهي قادرة على تحليل أنشطة الأندرويد واستخلاص معلومات قيمة عن كيفية استخدام أساليب رد الاتصال في دورة حياة التطبيق. تُظهر النتائج التي تم الحصول عليها ، أي أساليب رد الاتصال تم استخدامها في الغالب ، ولماذا استخدمت ، وما طبيعة التعليمات البرمجية التي تحتوي عليها. وبشكل أكثر تحديداً ، تظهر النتائج أن هنالك نسبة كبيرة من أساليب رد الإتصال تم استخدامها بشكل غير صحيح وأيضاً عدم الحصول على موارد النظام والإفراج عنها بشكل صحيح. وتشير النتائج أيضا إلى وجود جزء كبير نسبياً من مطوري برامج الأندرويد الذين لم يفهموا نموذج دورة الحياة بشكل جيد. وكما يقارن هذا البحث أيضاً النتائج التي تم الحصول عليها مع أفضل الممارسات وأحدث التقنيات المستخدمة في تطبيقات الأندرويد.

Acknowledgement

Thanks first to ALLAH for his guidance that without I will not achieve what I have done up to this moment.

I am deeply and forever indebted to my parents and my family for their love, support and encouragement throughout my entire life. I am also very grateful to my brothers and sisters.

I would like to express my sincere gratitude to my advisor Dr. Samer Zein for his continuous support of my master thesis. He was very patience, motivated and always provided his excellent guidance in helping me to write the thesis.

Thanks to the examination committee for their valuable remarks, especially Dr. Majdi Mafarja and Dr. Nariman Ammar.

Finally, I would like to thank my best Friends for their encouragement and support.

Contents

List of Figures

List of Tables

List of Abbreviations

OS	Operating System
UI	User Interface
SAALC	Statistical Analysis of Android Lifecycle
DVM	Dalvik Virtual Machine
SDK	Software Development Kit
APK	Android Package Kit
ALCI	Android Lifecycle Inspector
SAAF	Static Android Analysis Framework
ASEF	Android Security Evaluation Framework
TPR	True Positive Ratio
TNR	True Negative Ratio
VSM	Vector Space Model
CDG	Dependency Graph Component
BG	Behavior Graph
FPR	False Positive Rate
ACUP	API call Usage Pattern
AST	Abstract Syntax Tree
DI	Difference index
F-Droid	Free and Open Source Software applications for the Android
CSV	Comma Separated Values
BNS	Bi-Normal Separation
MI	Mutual Information
AHC	Agglomerative Hierarchical Clustering
CPPM	Contrast Permission Pattern Mining algorithm
SDG	System Dependency Graph

Dedication

To my parents who have dedicated their precious lives for our success
To my lovely my brothers and sisters
To my brothers and sisters
To my Supervisor Dr Samer Zein
To my Friends and Colleagues

Chapter 1

Introduction

1.1 Introduction and Motivation

Mobile applications (apps hereafter) usage has increased exponentially with millions of apps being available at the online stores [1]. Nowadays, users rely on mobile apps to deliver their daily tasks. Indeed, mobile apps cover various fields such as social, business, health, productivity and gaming to mention a few [2]. Moreover, mobile devices offer the same functionality as the PC through wireless, web browsers, video, and audio. At the same time, the mobile app development is not a trivial task and has its own challenges [3].

Android is a major vendor and one of the most popular mobile open source Operating System (OS) in the mobile app market [4]. The Industrial analysts expect that the Android platform will remain the dominant mobile vendor for the upcoming years. Google Play is the main online store providing Android apps. Since Android first release in 2008, developers have been heavily contributing in developing new apps that facilitate various user needs. As a result, in April 2017, the number of available Android apps has exceeded 2.8 millions [1, 5]. Further, the number of worldwide downloaded Android apps from Google play was estimated in billions in 2016 to

2017 [5]. Accordingly, the complexities of mobile apps increase to fulfill a variety of functionalities and features.

The mobile app development is typically different than other traditional web and desktop paradigms. Developers are facing a new set of challenges such as developing apps for different platforms (iOS and Android), and handling the issues of OS and hardware fragmentation, and lifecycle conformance [6, 7, 8, 9]. Even though much research has been directed to address these challenges, little research has been done in the area of lifecycle conformance.

When developing for Android, activities represent User Interface (UI) and each activity goes through different states during its lifecycle. These states are running, paused, stopped and shutdown. Each activity makes transitions between these states due to some events, such as receiving an incoming call, by calling a specific callback method [6]. Android developers need to have an appropriate understanding of the life cycle model in order to develop apps that function correctly [8, 9, 3]. Google documentation provides narrative information about the lifecycle model to assist developers in building robust apps [8]. However, a large fraction of Android developers is known to be novices and amateurs who may not follow the life cycle model and will end up with unreliable and faulty apps [3]. Additionally, there are little automated testing tools available for Android that enables to check the correctness of the app lifecycle [3].

This thesis aims to explore how android app developers actually utilize the lifecycle callback methods. More specifically, the aim to analyze Android open-source apps to reveal how these apps are built in terms of lifecycle callback methods and the utilization of system resources such as Camera, GPS, Sensors, etc. Analyzing Android app's source code is one of the most recent topics in the statistical analysis field [10] and provides actionable insight about how these apps are developed [11]. For

instance, it helps increase the quality of the code and improve reliability and performance of the software [11, 12]. Another example is rule mining [13]. Rule mining aims to extract hidden rules from existing project in order to improve new development projects [13]. Further, rule mining has been used in automated defect detection for complementing the compiler work and this is done through analyzing the source code to find the most common bugs [10, 13]. Indeed, analyzing the source code gives more insights and helps the research community and the software industry to understand how developers actually code their apps. In other cases, it can be useful to understand the architecture of the app and consequently to reduce the development time and programming effort [11, 13]. Other benefits of analyzing source codes include identification and elimination of security vulnerabilities in software [14], and provide statistical measures about the code such as numbers of methods, attributes, parameters, children, line of codes, depth of inheritance, complexity, couples and coherence [12].

Although there are a lot of studies in analyzing Android source code insights and patterns in different fields such as testing; quality; design; privacy and security; there has been no studies focusing on analyzing Android source code for lifecycle development. To address this need, an exploratory study was conducted to analyze Android source code. The main aim of our study is to explore how real Android developers develop their apps in terms of lifecycle callback methods. To achieve this, 842 open-source Android apps containing 5577 activities was analyzed. These apps were downloaded from F-Droid repository. Our dataset includes different apps with varying code sizes and from different categories such as Gaming, Navigation, Internet, Multimedia, etc. A statistical analysis tool called SAALC was built which is able to analyze and extract all data related to activity lifecycle callback methods. The resulting statistics reveal the usage of callback methods and where system's resources such

as Camera, Bluetooth, GPS, etc., are acquired and released among other important information. Also, the nature of code implemented in these callback methods was analyzed to understand for what they are used.

More specifically, the results show that onCreate() callback method is mostly utilized (92%) among all activities. On the other hand, the onRestart() and onStart() callback methods are about (1%) and (6%) respectively. Further, the onDestroy() and onStop() have (14%) and (6%) respectively. Also, the findings showed that the average percentages of wrongly acquired and released system resources are about (20%) and (8%). Such results enable us to understand more how Android developers utilize lifecycle callback methods.

1.2 Thesis Contribution

Analyzing Android's source code has been implemented to highlight the gap about the importance of the activity lifecycle in mobile apps. Our methodology of analyzing source code lifecycle patterns was applied to a real dataset of Android apps to generate statistics. To perform the analysis and gain results, an analysis tool called SAALC developed. SAALC was used to analyze 5577 Android activities extracted from 842 open source Android apps.

Accordingly, the focus was on exploring how Android developers utilize activity lifecycle callback methods, manage system's resources and the nature of code inside callback methods. Moreover, detailed comparisons were performed between different app categories to draw actionable insight and provide useful information about the most occurrence category. However, identified if any improvements or alterations are required to aid developers.

1.3 Aims and Objectives

1.3.1 Aims

Several analysis studies have been performed in the Android source code. However, these studies focused on finding code patterns in privacy, security, design, testing, energy, localization and prioritizing. Based on the literature review, there are no study focused on analysis Android lifecycle, except one study that focused on analysis Android for testing lifecycle conformance [3]. Consequently, this research aims to narrow this research gap and conduct a quantitative research. This research uses a statistical analysis of the Android source code. Analyzing Android activities source code helps to extract lifecycle useful patterns. The resulting patterns were used to provide general statistics and draw actionable insight about lifecycle utilization of community. Apart from that, Android developers can use our findings to gain insights into the Android app development. Further, researchers can use our findings to provide support for developers to overcome these inconstancy between Android model and documentations.

1.3.2 Research Objectives

Based on above aims, the following research objectives have been formulated:

- To develop a tool for analyzing Android activity lifecycle files.
- To generate and analyze statistics about callback methods utilization.
- To generate and analyze statistics about acquiring and releasing of system's resources during the lifecycle of Android apps.

- To generate and analyze statistics about the nature of code inside callback methods.

Based on above objectives, the following research questions (RQs) are formulated:

RQ1: To what extent Android developers utilize the lifecycle callback methods in developing mobile apps?.
RQ2: Did Android developer correctly acquire and release the Android system's resources?.
RQ3: What is the nature of code implemented inside onPause(), onStop, and onDestroy() callback methods?.

1.4 Overview of This Thesis

The remainder of this thesis was structured as follows. Chapter 2 presented a background. It showed Android platform, model lifecycle, and its challenges. In chapter 3 presented a literature review and discussed the pertinent literature and sources available in order to produce the thesis methodology. Then, chapter 4 showed the thesis methodology. It proposes a quantitative research. In chapter 5 showed the results of the thesis methodology using our development tool on the collected Android dataset. Chapter 6 presented a discussion of our results. Finally, chapter 7 provided a small conclusion about the research so far and the findings, and also showed threat to validity and future work.

Chapter 2

Background

This chapter presented a background on Android OS. It was arranged as follows. Section 2.1 showed an introduction about Android platform. Section 2.2 showed Android lifecycle model and callback methods.

2.1 Android Platform

Android is a Linux based Operating System (OS) which was designed and developed for mobile devices by the Open Handset Alliance in 2007 [15]. Android is an open source which deployed under the Apache License [8]. This helps developers in the development of the Android OS and its model lifecycle information.

In Android, the Linux kernel interacts with the device hardware, whereas the app APIs run after the kernel. Android contains a stack's layered software from app to API to OS to Linux kernel as shown in Figure 2.1. The Linux kernel called a hardware abstraction layer. It provides a process for memory management, security, and network models [16]. Further, There are some libraries run on the layer over the kernel such as SQLite, Webkit, and SSL to provide system functionalities.

All API assigned to the app framework layer to provide access to the device hardware. Each app on Android act as an interface to the APIs inside the API layer. It runs on Dalvik Virtual Machine (DVM) under a unique UNIX UID. These apps assign in the top layer [16]. Some of these apps are preinstallation as phone and home while the others were downloaded from Google play market to provide the user extend functionality [16].

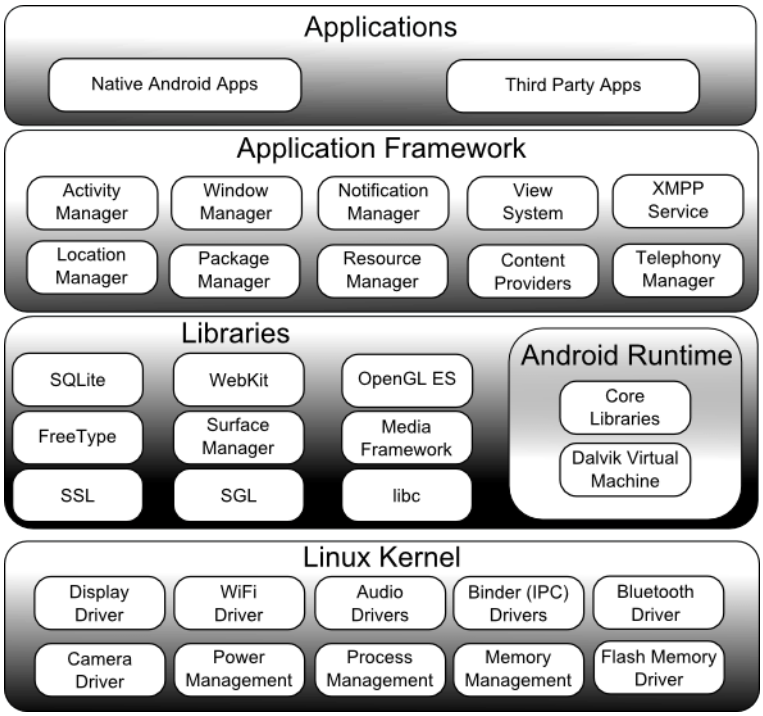


FIGURE 2.1: Android platform architecture

Android app is written in JAVA language [16]. The app codes compile using the Android Software Development Kit (SDK) tool which developed by Google into Android packages called Android Package Kit (APK) file. SDK also provides a testing environment using emulator features to allow developers test functionalities of apps using virtual environment [15]. However, when a user installs an app, Android device will install the APK apps file.

The Android packaged APK file contains a class .dex which a single file that holds a bytecode to be interpreted by DVM and other files in held in res/ and assets/ folders [16]. Also, it consists the Android manifest file at the root of the Android project directory. The Android manifest file is an XML format file type. Each app contains a set of components were declared In the manifest file. The Android system checks whether these components exist in the manifest file before running apps. These following components are:

- Activities: it represents a graphical UI of an app.
- Services: it is running in the background.
- Broadcast receivers.
- Content providers.
- App permissions.

An Android app consists of a set of activity component. Thus, Android provides some features concern to its activity's lifecycle. In this study, the main focus was on the Android activity lifecycle. However, activity's lifecycle was very extensively introduced in the next section.

2.2 Android's Activity Model Lifecycle

App's lifecycle refers to multiple concepts [8]. In this research, it does not refer to apps software engineering lifecycle which includes (requirement, design, coding, testing). Instead, App's lifecycle refers to multiple states and transitions. However, mobile app's lifecycle differs than other platform app lifecycle, such as the desktop or web.

In Android mobile platform, each app contains sub-functionalities which represent a set of activities [8, 9]. Moreover, each activity has its own lifecycle. The activity's

lifecycle can be defined as a set of states and the transitions between them during a runtime of Android's app. However, passing data between these states is implemented by sharing resources or asynchronous messages.

During the changing between states of an activity, events might occur [8, 9]. These events might cause in case of low memory, low battery, the triggered event from an external app such as incoming calls, or by the users themselves such as switching to another app. Particularly, on an Android device, Android's OS can't control the state of an app during lifecycles change due to limited resource and screen size. Thus, the OS must ensure efficiently the device resources. It may swaps out or kills an app without saving its current state In case of a lack of resources. However, Android Activity Manager inside Android OS is responsible for receiving and handling these events [9]. So that, handling events is done between the app and Android's system. Thus, the synchronous event handling is done using triggering lifecycle's model by the app itself.

Accordingly, developers themselves must control and ensure that no data is lost when the state changes when developing Android apps [8, 9]. The developers react the changing state and handling events by overwriting the callback methods such as onCreate(), onResume() and onPause()..etc. These callback methods executed when an app changes his state e.g turn video off when an incoming call event happens. So that, its responsibility for Android developers to override/implement the callback methods, in another word (lifecycle implementation /conformance).

What's more, a correct conformance of lifecycle in Android is very important to produce high robust and stable apps [8]. To understand that, imagine that during a user has typed a long message in Facebook messenger, the incoming call event occurs. Then, the event triggers OS to pause the currently opening messenger app and open

the phone app. During closing the messenger app, the callback method called onPause () is called by the OS. So that, if the developer doesn't save the message text inside onPause (), the message will lose and the user has to retype the message again after the call finish and resuming to the messenger app.

Another fact about the conformance lifecycles in mobile apps is that it is very stressed [8]. Different Android mobile devices like phone or tablet, has restricted user interface. That means only one app is visible on the device screen. Additionally, the mobile resources such as CPU, battery and memory are limited on these devices, so that the mobile platform schedules processes and makes the only visible app in the active state. This scheduling strategy will give only the visible and active app the required resources. By reflecting this fact, which related to the mobile platform over than other platform as the desktop, on lifecycle conformance that means for each time that a user returns to the home screen or switch between apps the callback methods are called because of a user can not open more one app in each time.

Android developers have to comply with the lifecycle model [9]. Android vendor offers a lifecycle model and documentation to help developers comply to activity's lifecycle [17]. Thus, developers must follow it [9]. In addition, the Google official website is the main portal for the Android's developer, it contains the guidelines for the Android's development model lifecycle and documentation [8, 9, 3].

The Android's activity lifecycle model is shown in Figure 2.2 from the Android developer's guide [17, 8]. It shows that an activity must be on one of five states. These states are [8]:

- Start state: When a user invokes an activity.
- Run state: When the activity stays in the foreground (full UI).
- Pause state: The foreground activity is partially viewed on screen.

  - When another activity obscures the running activity, lost user focuses or screen locks.
  - When the activity is in the run state, then the screen locking has been activated.

- Stop state: When the activity was not visible on screen. It is running in the background and remaining in the memory.
- shut down state: No activities exist in memory.
  - When Android OS kills the activity process which runs in the memory.
  - When Android OS kills the activity in the stop state in order to free resources.

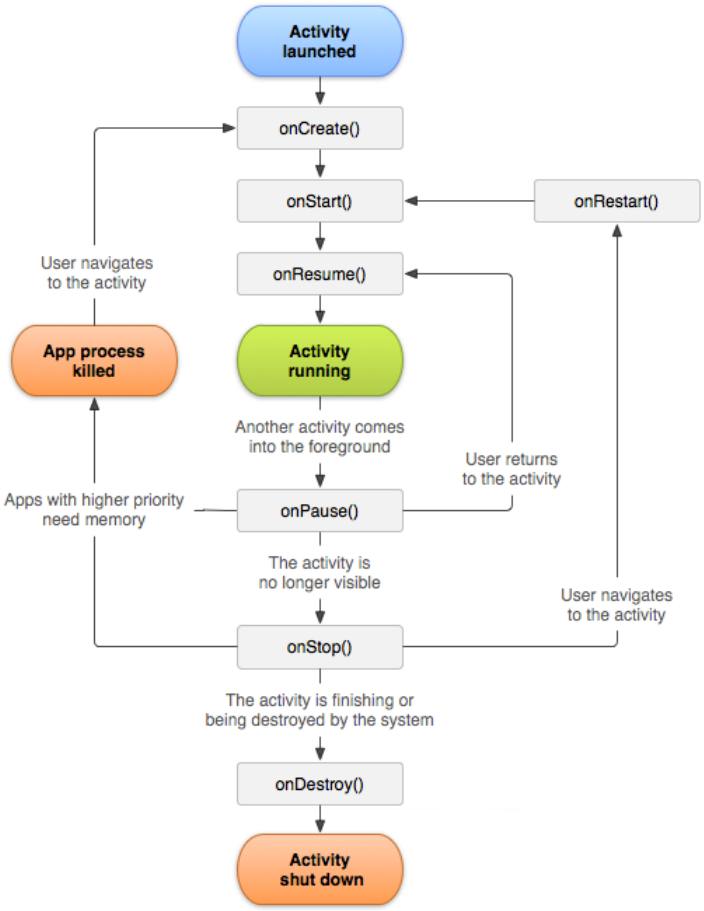


FIGURE 2.2: Android activity lifecycle model [8]

In Figure 2.2, The ellipse shape represents an activity state while the rectangle shape illustrates lifecycle's callback methods, which developers can override to react events over other state transition [8, 9]. The Android's model and documentation show seven callback methods and its detailed description as follows [17]:

- The onCreate(): It uses to initialize the activity. It also calls when the activity is created. Inside oncreate(), setContentView(int) is usually used to define UI, layout resources and views for the activity. In addition, the setContentView(int) is used to retrieve widgets from the layout. Some setups are done inside the

onCreate() such as bind data and provide a bundle to keep the previous state of the app. The onCreate() permanently followed by the onstart().

- The onStart(): It calls when the activity is becoming visible on the screen. It also followed by the onResume() if the activity was in the foreground or the onStop() if the activity was hidden. However, there haven't been more useful detail in how to use instant () in the Android's document.
- The onResume(): It calls when an activity starts interacting with a user. So, the activity will be on the top of the activity stack. Inside the onResume(), developers start initializing and acquired system resources and connections to networks and databases. It permanently followed by the onPause().
- The onPause(): It calls when an activity is becoming invisible or when a user is loing the focus on an app. It also uses to deal when a user leaves an activity. Any change made by a user should be committed in the onPause(). The data must be saved to persistent data or content provider to keep it from lost. Moreover, system's resources must be released or stopped animations and other things that may be consuming CPU. Some developers use onSaveInstanceState(Bundle) method instead of the onPause() to save any dynamic instance state in the activity into the bundle, to be the last received in the onCreate(Bundle) if the activity requires being recreated again. But this is a wrong because the onSaveInstanceState(Bundle) is not a part of Android callback method's documentation.

  An app enters the pause state briefly, so to keep the app performance and reliability, developers must avoid putting a heavy or long running code inside the onPause() callback method. The onPause() callback method followed by the onResume() if the activity gets back to the front and show in the foreground, or the onStop() if the activity becomes invisible to the user.
- The onStop(): It calls when the activity is invisible to a user because another

activity is becoming in the run state. Then, the activity becoming in the background and running in the memory. Developers must insert a heavy or long running code inside the onStop() rather than the onPause() such as closing networks, threading, and database connection. The onStop() callback method followed by the onRestart() if a user opens the activity again or the onDestroy() if the activity is killed by the system.

- The onDestroy(): It is the final call of the activity. It calls to perform any final cleanup when the activity finishes or when the system kills the activity to keep system's resources such as memory and CPU. However, developers mustn't use the onDestroy() to store data. Thus, if an activity modifies data in a content provider, These modifications must be committed on the onPause() due to there are some cases that the system kills the activity without calling the onDestroy() callback method. Instead, this method can be used to free resources such as threads.
- The onRestart(): It calls after the activity has been stopped, before to it being redisplayed again. It also uses to require a cursor object. Cursor object provides random read-write access and management to the result set that returned by a database query. The activities that using a cursor object instead of accessing and managing a database using managedQuery() require the cursor object again inside the onRestart(). This happens because these activities must deactivate the cursor object inside the onStop() callback method. Moreover, the onRestart() callback method also followed by the onStart().

Developers override these callback methods inside the activity class as shown in Figure 2.3.

However, In order to start an activity these cases might occur [8]:

```
public class Activity extends ApplicationContext {
    protected void onCreate(Bundle savedInstanceState);

    protected void onStart();

    protected void onRestart();

    protected void onResume();

    protected void onPause();

    protected void onStop();

    protected void onDestroy();
}
```

FIGURE 2.3: Overriding callback methods inside an activity

- The startActivity(Intent) method is called. It takes an intent as an argument, which describes the activity to be executed.
- When a user invokes an app, the onCreate(), onStart() and onResume() will be called and the activity's state will be changed to the run state.
- If another activity turns up at the front of the running activity, the onPause() callback method will be called and the app run state will be changed to the pause state.
- If the paused app becomes unavailable and the other activity becomes in the foreground, the onStop() callback methods will be called to change the app state from pause to stop state and the app will be run in the background inside a memory.
- If the stopped activity is invoked again and run in the foreground, the on-Restart(), onStart() and onResume() will call again to make the app again in the run state.
- If another app needs memory and system resources, the OS will destroys and kills the stopped app and the app will be in the shutdown state.

In addition, users may do some action that reflects the activity state:

- BACK button: When a user clicks on the BACK button, the system will kill the app and change its state to the shutdown state.
- HOME button: When a user clicks on the HOME button, then the system will call the onStop() callback methods. Then, the app remains running in memory and its state changes to stop state.
- Orientation: Android offers landscape or portrait orientations. When a user changes orientation between themes, the app will destroy and recreate again.

Android activity source code can be analyzed to find statistics related to activity's lifecycle development. These patterns are related to the callback methods usage, and other challenges that might influence activity's lifecycle development. The next Chapter 3 introduced the analyzing methods, lifecycle development challenges, and literature review related to analyzing Android source code.

Chapter 3

Literature Review

This Chapter presented a literature review in the area of analyzing Android source code in various sectors. The review was based on a selection of published literature predominantly in analyzing Android source code. The search period was set from 2010 to 2017.

The **keywords** used in the search: Android; Lifecycle;Static analysis; Mining;application; Apps; source code.

Deep search was run in Google Scholar, IEEE, springer, ACM and ICIT databases. Papers which discussed analyzing Android source code or Lifecycle topics were selected from the literature. The methodology of critical writing was followed to guideline based on the paper "how to or not to do literature review" [18].

In particular, this chapter showed the analyzing of Android source code in various fields such as design, privacy, security, quality, localization, prioritizing, energy and testing patterns. It also arranged as follows. In section 3.1, it showed an introduction about a current Android analyzing studies. Section 3.2 showed analyzing methods of Android source code. Section 3.3 showed lifecycle studies and challenges, whereas section 3.4 showed privacy and security analyzing patterns which was divided into permission, API, data flow and control flow patterns.

Section 3.5 showed design analyzing and section 3.6 quality analyzing patterns. Further, section 3.7 showed other analyzing patterns which includes a one studies of each related topics in testing, prioritizing, localization and energy analyzing patterns. After that, section 3.8 highlighted the thesis gap and section 3.9 provides a conclusion. In order to give readers a quick access for the literature review, the Figure at AppendixA.2 was represented to show a brief map for the literature review studies.

3.1 Introduction

Analyzing Android source code is used to find code patterns. Modern research in fields of analyzing Android source code patterns focus on different fields such as design, quality privacy, security and others. This thesis focus on a statistical analysis of Android apps for lifecycle coding patterns field. These are very important as it can reveal in depth insights of developers coding behaviors. As shown in this chapter, there has not been a previous research yet in the field of analyzing lifecycle coding patterns for a huge dataset of Android activities.

3.2 Analyzing of Android Source Code

Analyzing source code is not a new topic, it was used in different programming languages such as JAVA [19]. However, there are few research points to use analyzing in Android source code.

Analyzing source code includes several methods using a statistical or dynamic/mining analysis [19]. Both a statistical or mining analysis has no major difference. They work to find and extract hidden truths from a set of data. These extraction truths are called

patterns. A statistical analysis is a base of a mining. Moreover, both of them help to make decisions, learning from data and turns data into information.

Statistical analysis focused on quantifying data [19]. Using analysis tool, a relevant and hide properties and patterns of large data were extracted using a different type of statistics such as descriptive and inferential statistics. Whereas, mining focuses on finding a relationship and patterns on large data using different mining techniques such as classification, association, and visualization. Indeed, mining also includes an estimation and prediction theories and decides a noise from significant results.

A lot of analyzing studies has focused on statistical or mining was shown in the next sections of this chapter [10]. Further, a lot of tools were developed using one of these methods of analyzing. These tools performed analyzing of source code for different purposes. Some researchers used the analysis tools for code review and to help developers to check errors as compilers. The others implement it on the source code to extract hide patterns from code.

In this thesis, a statistical analyzing was implemented using our developed tool [19]. It depended on summaries and arranges data to draw conclusions from entire Android's activity source code dataset.

3.3 App Lifecycle Challenges

Understanding app lifecycle is the main point to produce more quality and reliability Android app. Actually, there were little number of studies that consider a lifecycle model in their research. This section showed three main studies related to the mobile lifecycle and its challenges. The first was [8] in reverse engineering and testing approaches. The second was [8] also used assertion test-based approach. Then, the

third was [3] which used static analysis approach for testing conformance of an app lifecycle.

Franke et al. [8] proposed a case study approach to reverse engineering of a form of dynamic analysis for the app lifecycle. Their approach was applied in four steps. The first was a full implementation of apps lifecycle. In this step, developers create apps follow the lifecycle model and overwrite callback methods. The second was log Injection by adding logging functionality to callback method. Logging includes callback method's name and the current app name. The third was transition trigger detection to get a list of triggers which causes the reaction of the android system. The fourth was an app Lifecycle model rebuild that provides developers a correct lifecycle model based on the collected information.

The findings of Franke et al. study argued that the Android activity lifecycle model and documentation which published by Android's vendors are informal, inconsistent and incorrect [8]. They have discovered errors in the transitions between the states in the official activity model, they have found the inconsistency between the official activity model and the Android documentation offered in Android developer's guideline. In the activity documentation, for example, the shutdown state did not define while the model representation includes the two run and shutdown states. Further, the pause and stop states did not represent in the model while they defined on the documentation.

An important point which also they found that the running apps in the foreground definition is incorrect. The app in the foreground means that it has user focus, but the researchers found that the running app could be without taking a full focus of a user. Consequently, the incompleteness and inconsistency in the activity's lifecycle model and documentation will cause some problems for developers during the development Android apps.

All these findings challenge lifecycle's conformance. So that, They also followed the approach of reverse engineering to the Android activity's lifecycle model to overcome these challenges [8]. They rebuild a new model which is correct, formal and consistent. The new Android's activity model is shown in Figure 3.1.

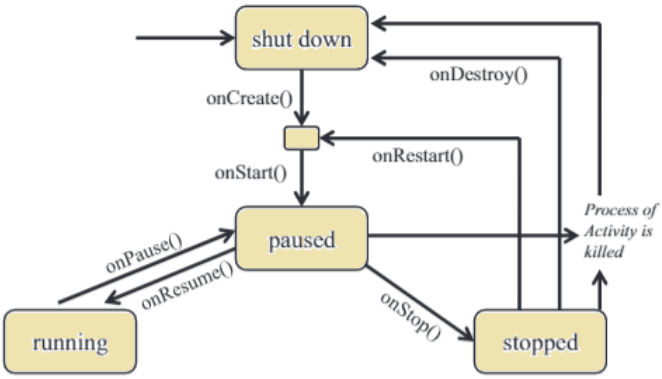


FIGURE 3.1: Android activity rebuild lifecycle model [8]

The rebuild model consists of four states [8]. These states are shutdown, pause, run and stop. Between shutdown and pause states, there is a small state which do not have a name because the activity passing through briefly. The transition between the states shows the callback methods which will be called to react events. The activity passes through the onCreate() or onRestart() briefly, then go to the onStart().

In their model, it shows that the activity on the run state will not be killed [8]. But, when an app in the pause or stop states, the system may kill the app without calling any callback methods. Thus, if developers store the data or close the opened connection during the onStop() callback method, the data might be lost. That leads to bugs and runtime errors whether the system will kill the activity. Instead, developers can store the data and close the connection inside the onPause() callback method. Because of that the onPause() is the first callback method that the system will call before killing the app. Whereas, developers must avoid defining these in the onPause()

method in case of a heavy code's functionality, due to causing the ineffective quality of the app. They can do it on the onStop() method instead.

The other points shown in the new model that the activity will remain briefly on the onCreate() or onRestart() methods, then will go to the onStart() [8]. So, developers must avoid establishing connections and networks during the onCreate() or onRestart() and do it in the onStart() instead. Also, the activity can briefly pass the onStop() then to the onStart() without calling the onResume() method. If developers close the connection during the onStop() and start it during the onResume(), This will cause run time errors because onResume() will not call.

Furthermore, in their findings argued that there are other challenges that might face the mobile's developers concern with to the activity's lifecycle which, was the ability to test the activity's lifecycle for each type of mobile platform versions is very difficult [8]. However, developers usually use the emulator or simulator to check that the app's functions achieved the user's perspective. In fact, they found that the emulator or simulator behave differently than the real device. So that, many events caused by lifecycle model can't be tested.

Again and due to testing the activity's lifecycle causes many challenges for developers as mentioned previously, there was another study of Android lifecycle. Regarding to [9] study, Franke et al. depended on the previous approach of reverse engineering for correct Android lifecycle model. Their study's approach followed assertion test-based approach to check the functionalists whose state or value might differ between lifecycle callback methods. It used to test data persistence of components such as text fields or databases, so if the current value of any component change after calling the callback method with the old value, that means it is wrong in lifecycle implementation and the assertion test will be fail.

In their approach also, they test a connection status, such as the internet or Bluetooth

and hardware status like Wifi and speaker [9]. The test scenarios were applied to Google Notepad app using three scenarios and different Android libraries. Then, a tool that supports the correct implementation of Android app's lifecycle was built through integrating a result of the lifecycle testing approach. The tool helps developers to insert assertion and execute a initialization step of testing app's activities. Using the tool also, a developer can decide a type of assertion, parameters and in which callback method the assertion will be involved. Further, this tool helps and reduces a development effort of testing lifecycle dependent properties.

However, another challenge related to activity's lifecycle is that developing a high quality, more reliable and robust mobile apps needs a full understanding the activity's lifecycle model, states, and transition as showed in [3]. Zein et al. argued that the task of development mobile app is more complex. Due to limited tools for the related issues of activity's lifecycle, which help developers in the lifecycle conformance, most bugs in mobile app development are related to lifecycle conformance. In their study also, they argued that developers need to know whether the system's resource such as *Camera, GPS, USB*..etc was acquired to use or released correctly during lifecycle activity's conformance. Also, developers should guarantee that the app manages system's resources and do not lose a data during changing the activity states. If the app can't acquire and release the resources correctly, this will lead to runtime errors. So that, developers should refer to Android developer guidelines which contain information which helps them to manage the system's resource. In their findings also showed the way's in how developers manage resources and loss the data also was affected by the inconsistent in the old activity lifecycle model. Although, the new model was considered on the official website. But, there were still a lot of novices and amateur developers who do not have a deep understanding of the activity's lifecycle models.

Leading to this challenge, Zein et al. presented a study related to lifecycle conformance which concern with the new Android correct lifecycle's model and rules by [8] and the result testing lifecycle scenarios by [9]. In their study showed that an app conforms to lifecycle rules if it's able reacting to a lifecycle state change well, storing data and managing system's resources such as *Camera, Microphone, Video, Network , and GPS* correctly. When an app paused or sent to the background, a resource should be released. However, failing in managing system resources consumes the main mobile resources as *battery, memory, and CPU* also causes runtime errors.

Also in their study, Zein et al. provided an automated approach depends on a static analysis tool called Android Lifecycle Inspector (ALCI) to manage a mobile system's resources during different lifecycle stages. ALCI helps novice developers building very qualified mobile apps. It is implemented using a mobile software model which extracts the lifecycle system resources rules and creates a repository for these resources. ALCI approach also analyzes the source code of an app against the rule lifecycle's models to verify that whether an app has been correctly initiated and released system resources.

ALCI's resources rules repository which contains formal, simple and complex rules for resources was built [8]. However, the lifecycle's rules change over time. So that, ALCI's model was built compatible with platform's versing. ALCI uses the generated model rule to match resources API call and decide whether the resource managed correctly. However, developers wrote codes in different patterns and styles. To overcome this, ALCI algorithm able to find two common coding patterns. The first pattern when the developer releases a resource inside a callback method. The second pattern, when a callback method calls a function that releases a resource.

The input of ALCI algorithm was the model resources rule with a list of system resources and an app source code [8]. While the output was a generated report contains

a list of warning about the resources that have not been acquired/released correctly. ALCI approach was evaluated into real Android open source apps depends on the seeding bugs principle. The evaluation considered two factors. The first was detected incorrect and correct releasing of system resources. And, the second was performed. The evaluation result of ALCI over ten apps showed that ALCI was able to static analysis of the system resources during lifecycle stages.

ALCI was the first tool that uses a static analysis to analyze the Android lifecycle [8]. However, there was no previous study used a static analysis approach for collecting statistics about lifecycle utilization. The strong point here that, Zein et al. study's methodology can be used to explore a statistic and patterns related to lifecycle and managing system's resources as showed in this study. To introduce these issues which were related to activity's lifecycle, this study was covered all these challenges to provide an exploratory study about how Android's developers used the lifecycle's states and managed the system's resources.

3.4 Analyzing for Privacy and Security Patterns

Before 2008, developers uploaded their app to markets without any prevention [15]. So, millions of a third party and low price apps offered to users. However, the increasing development of mobile causes spreading of unsafe or malware apps. Malware apps mostly download into devices without authorization. Also, they cause unexpected behavior without users awareness.

Further, the ability to modify on Android source code makes the malicious code easy to inject into apps [20]. It makes Android device susceptible to threats such as denial of service, buffer overflow, SQL injection, memory corruption, intercepting SMS and retrieving API's information attacks. Therefore, it's important to protect a mobile

device from malware apps and detect malicious behaviors and threats [21]. Malicious detection techniques are considered to do that.

play prevented any app contains malware contents to upload in its market by scanning apps using antivirus.The available apps on online market such as Google play are not protected. They add to users an ability to merge third party apps [22]. During 2012, Google Although, apps were tested before uploading into Google market to exclude any app which includes malicious activity, the process of revision apps still unclear. Also, it's not easy to define malware app from benign. However, most users depend on free apps which available in unofficial markets that may use repackaged app contains malware.

Using malware on a mobile app is attractive goals for attackers [22]. The reason for a malware development relies on gainful industries because the race of malware development is available to any user for making a new malware. In order to detect malware in mobile apps, developers are interesting, understanding and studying android platform natures and its app security [15]. This makes researchers have a big attentions on a privacy and security issues.

The privacy and security issues in mobile platform consider permissions. Permissions protect users from unexpected behavior [22]. It prevents app to access privileged resources, especially user privacy resources. An app request a permission from a user during the app installed. However, users mostly don't read required permission [15]. There are four types of permissions. Firstly, normal permissions that do not need user approval. Soundly, dangerous permissions that need to a user approval before the app installed. Thirdly, the signature permissions that do not need user approval or knowledge. Its use of the app signed with the certificate of the device manufacturer's. Fourthly, the signature system permissions use when app created by

various vendors, use image system or signed with the certificate of the device manufacturer's. API level uniquely identifies the framework API revision and permissions [22]. For instance, in Android manifest file, there are 151 system level permissions available and over 4,000 classes at the API level 21.

Another issue related to Android security and privacy is a flow of a sensitive data [22]. The sensors in mobile device lead to deal with sensitive's information. Additionally, users depend on the authentication process when uses the online payment for example.

Android platform prevents app to access hardware resource privileges, other apps on a device and sensitive information such as location and contacts [22]. It uses some permission mechanisms to let the device owner to accept that. When the owner installs an app on the device, all permissions which are decided by developers must be granted. However, developers need a deep knowledge about the required permissions which are implemented in the manifest file over deciding a suitable API that required to use device features.

Android documentation of permissions and API helps developers to use a correctly and minimal set of permissions and select APIs [22]. Misuse of permissions or use many permissions threaten the privacy and security issues. This leads to failure in a functionality of features. However, Android doesn't provide ways to recommend using suitable API to permissions. In order to solve this, many approaches used static or dynamic analysis of Android OS or frameworks to study that.

Some of proposed studies provides a security and privacy systems which follow different approach's [21]. Some of these systems depend on the Android permissions system, using requested critical permission that is performed by apps. This might be unsuitable because the permission request from an app in some cases does not use the code but its need the package of advertisement. Whereas, some malware

doesn't want any permission. So that, other systems based on the bytecode semantic information. They depend on data flow analysis to check vulnerabilities and severe permission. By contract, these systems have the ability to check only specific vulnerabilities in the apps. However, some systems use API analysis, which overcomes and replaced previous systems.

This section shows some of these studies related to analyzing security and privacy patterns. These study related to analyzing Android apps concerns with previous security and privacy systems. This section included analyzing permission, permission and API, data flow and control flow techniques.

3.4.1 Permissions Patterns

Some studies used to analyze of the Android permissions system to find security and privacy concern issues. This section discussed four major systems working on analysis the permissions patterns.

The first study [23] proposed an approach called Mobsafe using analyzing, and cloud computing to evaluate mobile app security and to decide the malware and benign mobile apps. In other hands, the second study [20] presented an approach called Droid Permission Miner using static analysis to identify the permissions that lead to malware in mobile features using Androguard tool. Whereas, the third study [15] proposed a pattern analysis approach to identify a record and a used permission to differentiate malware from safe apps. Finally, the fourth study [24] presented analyzing techniques to seven thousand apps to decide how the permissions principle of used and how it interacts with other also how developer used Android libraries in the general market.

Xu et al. [23] depended on a cloud stack infrastructure and Hadoop Storage of a computing platform called Mobsafe. Mobsafe analyzed APK file from apps and stores result in the Hadoop Storage. Mobsafe also used a static analysis through adapting a Static Android Analysis Framework (SAAF) and dynamic analysis through adapting an Android Security Evaluation Framework (ASEF) in order to evaluate apps and calculate a total time for evaluation all apps in the markets.

ASEF used to analyze Android apps. After uploading an unknown app on Mobsafe, the logging and traffic sniffing was turned on [23]. Then, the app installed on the Android virtual machine. ASEF simulated the human interaction on the app and then compared the log of a virtual machine with CEV library and internet activities. Then, it analyzed the log file and internet traffics to check malware and vulnerabilities in the app. Further, the SAAF extracted APK file and analyzed the permission and patterns of the app. Finally, Mobsafe evaluated by collecting the dataset from the China app market.

The results of their study showed that, ASEF needed around 2 minutes to analyze 20 android apps [23]. By contrast, 33.93 seconds needed in the same apps using SAAF. Moreover, Mobsafe approach was practical to detect malicious apps from all stored apps in the markets.

However, Aswini et al. [20] depended on a huge dataset from Contagiodump repository. Around 209 malware and 227 benign Android apps from an open internet source were collected. Each sample set of malware and benign apps were divided into two parts. One half of a training set and one half of a test set. From the training set of malware and benign apps, permissions were extracted. In order to extract permissions from the sample, APK file was entered into Androguard. Androguard used a python script called androaxml.py to extract the manifest file from the samples.

The resulted permissions around 158 benign and 141 malware were extracted [20].

Then, filter process was implemented over the resulted permissions to extract features. After the filtering process, the number of feature sets was 63 for benign and 56 for malware permissions. Then, the permissions were grouped into four parts. There were 75 features are union malware and benign features, 44 common to malware and benign features, 12 discriminant malware features and 19 discriminant benign features. On the 44 common features, some features were excluded and pruned because it does not contribute to deciding the target class. After pruning, the including common feature numbers were 18. Moreover, the 75 union feature set was split into two target groups which are 49 benign and 26 malware.

Consequently, the feature selection of variable feature length was applied using Bi-Normal Separation (BNS) and Mutual Information (MI) on the 44 common features [20]. BNS was computed by the absolute difference between the normal cumulative distribution function of the true positive rate features and the inverse of the normal distribution cumulative function of the false positive rate features. Whereas, the MI was computed the dependence of two variables. In this study, The features of greater BNS values were selected.

The result of their study found that the random forest was the best classifier than a neighborhood [20]. Moreover, the top feature accuracy was lower than bottom features of the top and bottom features of BNS values. The comparison between BMS and MI feature selection to identify unknown samples found that MI was better. As a result, the MI with a small feature length was better classification accuracy.

However, Moonsamy et al. [15] depended on two different malware and safe app were collected. Each dataset was separated out two parts, one part for the used permissions and anthers for required permissions. So, the fours dataset was used in statistical analysis using frequency counting to define the most popularity permissions in each set. As a result, around fourteen thousand's permissions for malware were

founded across four thousand for safe. From these permissions, unique and common permissions were extracted.

Unique permissions which called for all permissions founded in each separate set, whereas the most common permissions which called for all permissions found in both sets [15]. Around 33 and 20 unique required permissions for safe and malware apps, 70 common required permissions and 5 unused required permissions were discovered. Whereas, 9 and 4 unique used permissions for safe and malware apps, 28 common used permissions and 87 unused used permissions were discovered.

What's more, a hierarchical Biclustering was applied using Agglomerative Hierarchical Clustering (AHC) on the datasets to offer figures and graphs that view the data distribution of the groups of apps which used a request or used permissions [15]. AHC applied to the row, column dimensional data matrix, and bottom-up clustering to find subclusters in a level tree form. The low-level clusters joined to produce new clusters. This process of joining clusters was repeated until joining all clusters of one large single cluster. Then, The use and request matrices were generated.

As a result, the required permission was observed more than used permission [15]. Also, the abnormal apps used a request or used permissions more than the normal apps. In the detection process between normal and abnormal app, the unique permissions were better than common permissions. After that, Contrast Permission Pattern Mining algorithm (CPPM) was implemented to define a set of permissions which utilized to differentiate malware from a killer app and to develop permission detection techniques. Further, CPPM worked on multiple datasets to find frequent, infrequent and combination permissions between them. It finds the candidate permission combination of each set using Apriori-based itemset approach. Then, it used Support-based candidate pruning used to calculate the frequency of a certain itemset on the resulted candidate permissions itemsets from the previous step. So, only the frequent

permission used in mining permission pattern because of its hold many features than infrequent permission. The selection of permission pattern related to support values of both datasets. Constant permission pattern had the greatest support value.

A result of CPPM in their study showed that, 23 permits were founded [15]. There were 6,15,1 permissions belonging to normal dangerous, signature and signature system category. Moreover, 56,31 required and used permission patterns was founded from malware dataset, whereas 17, 9 from a safe data sheet. The most required and used permission, from a dataset, was founded on the internet. The experiment result showed that the used permissions consider helping in the malware app detection, by contract the official documentation let some API level 3 utilized permissions that did not declare in the Android manifest file.

Moreover, Dering et al. [24] used 213 permissions from android documentation to decide how the permissions were used. They analyzed the manifest file and finding the similarities with the traditional permission was applied. The Jaccard which equal to the intersection of similar permission over all permissions for the result set and old set was computed. The Jaccard values are over 0.1 for permissions considered to select the similar permission. 55 permits were reminded. Then, a graph was used to represent the relation between resulting permissions. The graph showed the resulting permissions as a group of related functionality and tells about how to use these permits. However, to decide how the library was used, 100 common library namespace was used and grouped according to service. Seven thousand Android apps were analyzed and scanned the namespaces also decomposed them with the 100 common libraries. For each remixed library Jaccard was computed with threshold value over 0.25, then the graph was used to represent libraries set to find a cluster. Through the interaction of the libraries in a graph, developers understood how to build the apps and how to use library functionality.

3.4.2 Permissions and APIs Patterns

Some studies used Android permissions system and API feature's call to find security and privacy concern issues. This section discussed three major systems working on analysis the permissions and API patterns. The first study [16] showed an approach to classifying android apps whether malicious or benign using Bayesian classification techniques. Bayesian classification use frequently features, characteristics to detect malware and reduce them from apps market. The second study [25] showed an approach to evaluate clustering techniques using K-means algorithm on Android apps and apply them in malicious detection. In addition, the third study [22] presented AP-Miner approach using static analysis and data mining. It depended on the frequent API and permission used in existing apps in the market stores. Thus, AP-Miner helps to recommend for each API a suitable permit must be used, so it can assist in development when developing a new app.

Yerima et al. [16] study purposed a reverse engineering tool to disassemble.dex file into smaller files to extract features from APK file. Moreover, it generated features from the manifest file. The features set include API call, command and permission detection. Their approach designed in two stages. The learning stage which used a set of a malware sample. And, the detection stage, which extracted a feature set from the app to use them for detection. To increase the performance of Bayesian classifier, the feature set was ranked and selected the most relevant one in the feature selection function. One thousand malware and another one thousand benign sample app were collected to evaluate Bayesian classification using accuracy and error rate metrics. Finally, the result in their study argued that the approach achieves better detection rate and it's suitable to use Bayesian classification in the malware detection's.

Whereas, Samra et al. [25] study implemented through collecting android apps, extracting apps, processing and extracting data from the manifest files and creating an

ARFF file which contained a set of features. The ARFF file was entered to K-means algorithm in WEKA tool which classified the document to k-number cluster then give the evaluation. Their approach was evaluated by extract the features and requested permissions of eighteen thousand Android manifest files from business and tools category. The evaluation depended on the precision metrics which decides how many apps on a correct cluster of all clusters sizes, the recall metrics which decided the number of apps on the correct cluster from all apps and f-measure metrics which was a combination of precision and recall value. The three metric's value results equal to 0.71. The results in their study argued that the machine learning and clustering techniques were able to use in order to detect the malware.

However, according to Karim et al. [22] study around 600 apps from F-droid dataset were collected. Then, AP-Miner was used to analyze each app and extract the permissions and API from APK files. After that, association rules and frequent itemset mechanism were used to find the rules for these permissions and API. However, in order to find transition set of association rules, two types of mapping were implemented in the apps dataset. These are baseminer and filteredminer. In the baseminer mapping, itemsets were the apps dataset and frequent itemset was permitted and API used in an app. The API was founded on the import statement of Java source code by applying static analysis using SRCML. Whereas permissions were founded by passing the manifest file and extracted uses tags. Therefore, the training set of permissions and APIs was extracted. Then, using the filteredminer, refinement the training set was applied to remove unnecessary APIs and permissions by check the traceability of these. Lastly, the filtered miner mapping process was implemented on the two sets.

The results of their mapping were the groups of traceable permissions and APIs. Each group, the relations between the subset were considered. However, It seemed many

to many relations. So that, the equivalent groups were generated from these relations as transition sets. By applying association rules on a transition of permissions and APIs, frequent itemsets were found, according to the minimum itemset support value and confidence value which computed by the ratio of the API support to the permission support. The minimum support threshold was defined equal to 1 and the minimum confidence value of 40% to 80%. Further, the result list of association rules were defined as API class permission, then ranked according to the confidence and support values. Through resulting rules, AP-Miner recommended permissions for the API. These rules were compared with another recommendation as Androguard and sort according to some metrics. Androguard detected the traceability between API and permission and use a static analysis to map between permission and API.

The metrics were the precision metric which was measured by dividing the correct number of predictions set over a correct place in the correct number of prediction set; the recall metric which was measured by dividing the correct number of predictions set over the all number of the sets; and f-Score metric which depended on Recall and Precision values. However, The dataset was divided into 10 folds of 60 apps. For each 10 running, the metrics were computed. As a result, AP-Miner was recorded better precision and recalls than Androguard and Scot.

In the same study [22], another qualitative analysis was applied to check if the resulted association rules exist in the Android documentation. After Android documentation was analyzed and compared with a random sample of 30 rules of top confidence value. The result showed that AP-Miner was very accurate at the recommendation the permission to APIs. So, AP-Miner helped developers because it giveed a recommendation that not found in Android documentation. So, the resulting rules can use to update the Android documentation permission.

3.4.3 Data Flow and APIs Patterns

Some studies used android data flow analysis to find security and privacy concern issues. This section discussed three major systems working on analysis the data flow patterns. The first study [26] developed an approach called Androidleaks which is a static analysis, an automated framework that discovered a potential privacy leak. Androidleaks classified the apps to malicious leaks or benign leaks depends on the personal information that was transferred off in mobile. The second study [21] proposed a system which relies on analyzing and classification in order to replace the previous system. The classifier was built in order to detect malicious apps, identify malware patterns and take a suitable design to protect devices. it depended on the API level information which helps to recognize a malware from a benign app. Also, the third study [27] showed an approach called Mudflow which used a static analysis to define the flows of the sensitive data source in the mobile device. Mudflow used mining and classification to find patterns for the flow of benign app behavior. It compared the malicious and benign app depending on the treating of a flow of the sensitive data. Then, Mudflow utilized the patterns to detect malicious app behavior using the classifiers.

Gibler et al. [26] used Androidleaks to analyze JAVA code and bytecode depending on DED and dex2jar. The permissions were mapped with its API Which defined in the manifest. They also argued that their study helps to understand a suitable permission for functionalists. Through the mapping, the data flow between source and sinks was decided. Then, using WELA, a call graph and data flows were generated and analyzed to decide the potential paths which passed the private and sensitive data over the internet. Further, WELA produced System Dependency Graph (SDG) to decide the dependence on intra control and data in order to provide a taint analysis. Also, forward slicing was computed using the return value of the source method in SDG.

Then, the slice was analyzed to decide potential leaks of the sensitive data. However, WELA cannot handle the callbacks in apps. So, the listener registers method was decided during the mining code process. The type of listeners was defined for the method parameters, then the callback parameters were analyzed.

Consequently, in their study, around 24,350 Android apps were collected to evaluate Androidleaks framework [26]. Moreover, 2,342 apps were manually verified for some privacy leaks such as data, wifi, phone information, GPS location, and audio recorded with the microphone. So, their study result showed that the number of discovering leaks in 7,414 Android apps was 57,299.

Nevertheless, Aafer et al. [21] approach followed three phases. Firstly, the feature extraction phase, which was for extracting the most malware features. The data flow analysis approach and the extraction of APL level were used in the extraction phase. To apply that, the large sample apps around twenty thousand sets of malware and benign was collected from McAfee and Android malware genome project resources to generate its API call. The from the bytecode; the API level information such as the dangerous API calls; the package level information about invoking the dangerous API and parameters level information to passing parameter values when the API invoked; the app requested permission; particularly class; method name; the caller's package name; and the callee parameters were extracted.

Secondly, it was the feature refinement phase [21]. In refinement phase, the most common package as Android and JAVA packages were removed from the set, whereas the other has remained. Further, API calls which used in advertisement package or third-party package, which are responsible for the most malware in apps were extracted from the remaining set. Then, the itemset of advertisement, web tracking and web ranking were tested and compiled.

As a result, from the refinement phase, approximately 412 advertisement packages

were found [21]. Moreover, any API was invoked through the advertisement package was defined. In other hands from the feature set, the server input was defined using the data flow analysis. Also, the most frequent parameters were produced to decide the most severe parameters and each app requested them. The resulting parameters set was very large, so the parameter feature set was reduced and categorized. Then, the frequent API on malware rather than benign was defined. Moreover, from the APK files, the list of feature vector was created to link with the class names even malware nor benign. However, in order to apply the previous two phases, the Droidapiminer tool was built. Droidapiminer was a static analysis tool for python,it analyzed the API level to help understanding the malware activities.

Thirdly, the model learning and generation phase [21]. In generation phase, a rapid miner was built to produce the classification model. Four classification algorithms were generated from different types. The SVM algorithm relaid on learning method, KNN algorithm relied on lazy classification and the decision tree classification algorithms were ID3 DT and C4.5 DT. Then, the split validation was used in the dataset by dividing the dataset into two parts. One for training the classification models and the other for testing them. However, to decide the performance of each classifier algorithm, the accuracy test and evaluation were computed by dividing the number of malware and benign feature by all datasets. Additionally, the True Positive Ratio (TPR) and True Negative Ratio (TNR) were measured. TPR computes the percentage of the correctly classified malware apps, whereas TNR computes the percentage of the correctly classified benign apps.

However, in their study two experiments were implemented [21]. The first experiment was done with permission requested. The permission requested was extracted from the dataset and sorted according to a various usage. The top frequently malware permission requested set was considered to apply an accuracy test, TPR and

TNR. After the first experiment was tested, the result showed that the number of frequently set in malware was 64. So, if the set contained more permissions, the accuracy increased. The result of the first experiment concluded that the permission model was not robust because the manifest files malware authors can define benign permissions to fault the classifier. Moreover, the repackaged Android malware which seemed benign app and tried to achieve malicious goals cannot be classified.

The second experiment was done with the package and information level [21]. Using the fetcher vectors which included the most frequently API that generated before. The top 169 frequently API was classified and evaluated its performance. The result of the second experiment showed that KNN. C4.5 accomplished a higher accuracy and less performance classifier. Also, in the main dataset, the parameter model was generated and evaluated in the same way. So, 20 parameters were added to the 169 top frequently app. Finally, their study result showed that the KNN classifier accomplished higher accuracy and performance than others. As a result, the API feature's model, package, and the parameter level information was better than the permission model. Also, the best classifier model was KNN, then ID3, SVM C4.5.

According to Avdiienko et al. [27], around two thousand android benign apps were collected and fifteen thousand malware apps from Google Play. In their study argued that Mudflow helped to detect attacks and recognize a known attack. Also, it was useful for the user to understand the behavior of the apps of the sensitive data. For the sensitive data on Android, malicious apps used the sinks of the data source otherwise benign app. In Android, apps used to access the sensitive data. The abnormality of the malicious app detected using its flow of treating. The data flow analysis depended on a taint analysis to detect apps behavior. The taint analysis determined if the flow of the data comes from source to unwished for sinks such as sending information to the third party server.

In their approach Mudflow used Flowdroid tool to offer a taint analysis [27]. Also, Flowdroid gave the models of the interaction the operating system with the app life-cycle. Its extracted the all sensitive data flows from source to sink to generate a set of pairs. For each benign app in the dataset, the vector of probabilities according to SUSI categories was defined. The SVM used these vectors for classification the benign apps from malware. So that, their finding shows that Mudflow showed that the number of detected malware apps was around 86.4% also the 10,552 malware apps approximately 90.1% of malware app hacking the sensitive data.

3.4.4 Control Flow Patterns

Some studies used android control flow to find security and privacy concern issues. This section discussed two major systems working on analysis the control flow patterns. The first study [28] presented model called DENDROID to analyze malware and families in a mobile device using text analyzing for the code structures. DENDROID approach used a code structure which was not used before. It differed due to some reason. First, using code structures to represent the malware and families do strongly avoid obfuscation. Second, dealing with the huge amount of data becomes easily using analyzing. On the other hand, the second study [29] developed a system called Droidminer. Droidminer depended on static analysis and data mining of Android malicious apps to find malware patterns in order to detect malware on unknown applications. Moreover, for each detection, Droidminer decided the malicious families classification and its characteristics. Droidminer was designed to fit the changes in the codes. It was very useful in detecting the emerging attacks. Furthermore, it was automating, learning and updating the patterns by using new malicious samples.

According to Suarez-Tangil et al. [28] approach, Android malware genome project was selected to collect a dataset. The dataset contains around one thousand malicious app includes 33 different families. For each app in the dataset, decomposition the source code into code chunks which represented a method in class was implemented using Androgurad. Then, CFG was used to represent the sequences in each coding structure. Followed by applying text mining on the CFG. Further, around eighty thousand code structures was found in the dataset. Some mining definition was used to analyze such as variance, redundancy, size of the set according to families. Then, distribution the code structures according to its families were processed to find the frequency and the fraction the apps in the families. Also, the feature vector for each malware sample and families was decided by the Vector Space Model (VSM) which was recited by applying the computing family algorithm to the structure code. The nearest neighbor (1-NN) classifier was applied to the families of unknown malware samples depending on these code structures to find the family of each malware instance. Then, clustering was used in the malware families and analyzed using dendrograms (Phylogenetic Trees for Biological Species) to find the similarities and differences between each family.

Whereas, Yang et al. [29] used Droidminer to extract all sup paths and sub-sequences of malware control flow instead of depends on the method, class or field names detect malware. Then, 24,66 malicious apps were collected to evaluate Droidminer. Droidminer process defined in two phases. The first was mine and the second was identical. In the mining phase, Droidminer mined malicious patterns from a set of Android apps. The same malicious apps families have similar functionalities and behaviors. Droidminer used two tried graphs to represent the app's resources, API, and program logic in two layers. The first layer graphically represented by the Dependency Graph

Component (CDG), which demonstrated all interactions between the different component an app. Then In the graph of the second layer, Droidminer taken an app as input to generate the Behavior Graph (BG) using the Behavior Graph Generator component (CBG) which used to represent the functionalities and behaviors for the app components. Through the common behavior was extracted as patterns (API pattern resources patterns, control flow patterns) by mining all paths and subpaths. Also, it generated all sub-segments from the subparts.

Droidminer implemented the machine learning mechanism over the resulting set of patterns [29]. In their study, the four different machine learning classifiers (Naive Bayes, SVM, Decision Tree and Random Forest) were used to evaluate Droidminer. For each classifier's, the performance metrics as a False Positive Rate (FPR), detection rate and accuracy were computed. The result showed that Droidminer accomplished 95.3% of detection the malware rate by using Random Forest classifiers. Droidminer also saved the detection rate more than 86% for all classifiers. Also, Droidminer accomplished higher detection rate and lower false positive than using extracted permeation approach.

On the other hand, in the identification phase, Droidminer identified whether the app was malware or benign through generating the modality vectors from unknown apps and comparing them with the patterns from the first step [29]. So, the apps considered as a malicious if its malware patterns induction overran malware thresholds. A similar family classification for the mine phase was decided. So, the random forest classifier was used to evaluate the ability of Droidminer to decide the malicious family. A set of samples from 12 different families was collected for evaluation. Droidminer produced 92.07% accuracy classification. Moreover, Droidminer used the association rules to decide the malicious behavior characteristics from analyzing the patterns in knowing family classification relationships to decide malware behavior.

3.5 Analyzing for Design Patterns

Design in software engineering is an activity which starts with the ideas of the engineers and transform it into different models and diagrams [30]. Using modeling and design, engineers try satisfying customer's goals. For a long time, models, diagrams, and design patterns were involved in different studies. Design patterns are a general solution to a common problem. It may evolve over time. When a new better design pattern has developed, It may replace with more popular one.

On Android platform, there were few studies about design issues and patterns. In most cases, these studies depended on analyzing android source code to gain design patterns. Moreover, these studies were implemented methodologies that have been applied in the JAVA platform. In this section, two main studies discussed the design issues concern with Android analysis. The first study was about analyzing the changes of design patterns [30]. Whereas the second was about analyzing API call Usage Pattern (ACUP) [4].

Alharbi and Yeh [30] presented an approach to analyze Android apps to find evolves and changes in user interface design patterns using mining techniques. Their approach was used a study methodology that includes five main stages which called collect, decompile, extract, state and diff stages. These stages were applied by implementing a system that contains five components.

The first component was the apps and listing detail web crawler for collecting stage to download large numbers of the Android app and its updates and save its APK's file. Second, using the actual component to decompile stage to extract the app's codes and get a directory file tree in order to analyze UI design patterns. Third, the feature extractors component used to extract the source code and directory file tree features and save them in a data store. From these features, the manifest file, layout, and

string definition files were parsed. Also, some information related to a GUI structure was gathered. Moreover, a bytecode was analyzed to extract GUI behavior. The fourth components was the database client driver which was implemented as data store used to query and retrieve data from the database and gived outputs as CSV file. Fifth, the transformation trackers state stage component to calculate some statistics, complex and descriptive statistics. In the final, the data stage to compare the differences, common change and updates for GUI patterns in two different app versions.

Moreover in [30] study, the main method was followed for all findings depended on retrieving data into the database by applying multiples query structure depends on a pattern and the most related keyword in order to obtain. The methodology of their study was applied to the most eight design patterns. The result of their study argues that some app updates his detail to follow the design pattern changes. What is more, the others, which have not applied design patterns, try to switch to use a design pattern.

On the other hand, Lamba et al. [4] showed the developer styles and feedback of using Android platforms in mobile app for new development through analyzing and mining android apps in order to find API call usage patterns. ACUP "is a pattern of method calls, wherein all the methods are invoked together in the same user-defined method". ACUP, it mainly can define the error location and recognize functionality and software component influences. Which is why, their study stated the extracting a main popular invoked the ACUP, classes, methods, packages, and interfaces are more beneficial for the API consumers and producer relations. Moreover, it helped to maintain the level of development in android systems.

The main motivation in their study [4] that it had not been done previously for Android. So, its methodology depended on related work of analyzing ACUP for JAVA.

Also, it depended on the most popular methods, classes, interfaces and packages in JAVA. Additionally, it was the first study, which used graphically present it's finding result on API usage by advanced visualizations such as radar char, heat-map, bubble chart.

So that, thousands of Android source code apps were manually downloaded from F-Droid to implement their methodology [4]. At that time, F-Droid contained 13 categories for app also the whole of its apps provided in the Google play. For each app in the dataset, theAbstract Syntax Tree (AST) was translated from JAVA source code file. Then, a note was made for each method declaration also the invoked methods of a method declaration. The name of the method was used to represent AST transition. The transition was recorded by a set of invocation method which used to generate frequent itemsets. Then, analyzing and mining techniques was performed on the patterns itemsets. Further, SPME which is open source data mining tool was used in an analysis. Also, the Aproriclose algorithm was applied for mining the itemsets in order to avoid redundant patterns and find frequent item sets. The frequent itemsets were filtered based on the size. So, the patterns of size less than 4 was removed and did not consider as an ACUP. Consequently, The result showed that the most popular ACUP is on alertdialoginandroid. Moreover, the merging iterative process was applied using a coefficient index to compare a similarity and diversity of itemsets and help to present broad functionality.

The popularity of packages, classes, and interfaces were computed through counting the number structural unit that was imported to explore the relationship between Android platform and mobile apps [4]. That's helped API producers use better popular alternatives in order to enhance the quality and usability of the app. In other hands, the framework changes were measured to help API consumers understanding how much adaptation. Also, it helped API producer offering solutions to problems in case

of API changes that violate compatibility with older apps. Secondly, the most popular API changes were discovered. APIs set of specification differences which were above of level 10 was analyzed and put a note of modified elements in each version. In the results, Android MTP package was found which is the most used and introduced packages. However, The popular method was decided by computing the number of invocations. After implementing the result found getstring, tostring and add are the most popular method. The study was discovered that if any class is removed from an API, the API producers taken a lot of time to edit app's code.

Additionally, ACUPs was analyzed based on their occurrences of different categories in the dataset [4]. Their approach argued that API consumers introduced to know their interest elements in the app's category in order to enhance them in different apps. Apart from that, API producers understood how API implements in various app's categories. The ACUP that discovered in 9 out of 12 categories was viewable on Android, which concerned about the size, padding, and margins the UI.

The two previous studies [30] and [4] used data analysis and mining approach to analyze large-scale Android apps. All of them analyze some related design patterns. In [4] research focused on discovering ACUP to enhance developers for using related and popular ACUP, method, class, packages that help to increase the quality and usability of apps in order to raise the scale of customer satisfaction. While in [30] research focused on discovering the most popular GUI design patterns and the changes that will lead to creating more reliability and usability apps for customers.

However, both of them use different methodology to reach the study's goals. But, the main common point in each of them is popularity. So, in their depend on analyzing and mining to extract popular patterns of source codes, system files or some descriptive information by analyzing the number of occurrences of the patterns using name declarations or known keywords.

3.6 Analyzing for Quality Patterns

There are millions of Android apps in online markets. Quality is the most important issue that takes user attention in these apps. Markets provides a market rating system to offer a user ability of feedback an app. When Users install an app from a market, they will take into consideration the app rates in the market. So, developers try to develop high-quality apps to satisfy users' needs. However, developers cannot decide a quality of an app before publishing it to a market. In this section showed two studies about analyzing quality patterns of Android apps. Both of the two studies depended on analyzing android source code to gain quality issues. The first study was about app quality metrics, whereas the second focused on a quality of apps performance and GUI.

Shaw et al. [31] presented an approach that depended on analyzing and mining Android apps to decide a quality metrics to help developers. Then, around tens of thousand Android's APK files were collected from Slide Me market to process a reverse engineering using apktool in order to produce the bytecode with resources and manifest files. From these apps, the top (5.0) and lowest (0.5-2) 1000 apps rate was compared according to the average values of quality metrics.

Moreover, three quality metrics were used in [31] study. First was the size metric which includes the number of instruction, classes, method, method per class, instruction per method and the Cyclomatic Complexity. Second was the object-oriented metric which includes the number of children, depth of inheritance tree, access to public data, cohesion and coupling. The third was the Android specific metric which includes Unchecked bundles, APK file size, number of string resource and bad smell methods that may lead to bugs such as a show() and onkeydown (). Further, to gain a result from their study, the Difference index (DI) was computed to indicate which

the metric is a concern to the user rating. For each metric, DI equals to the percentage to the top over the bottom apps. Finally, the result of their study argued that Android specific's metrics are more suitable than other metrics for assessing the quality of Android apps.

However, Gómez et al. [32] presented a DUNE context-aware approach to detect a defect of UI performance that helped the developers to build more quality apps and decide the performance defect in new releases of the app. DUNE had four phases. The first phase was the gathering performance metrics. Performance metrics were analyzed for the frame and UI event statistics. The second was an assembling data which can be used to build a model in order to detect a performance defect. The third was the detect performance deviation by comparing the app defect with the performance metrics. And, the fourth was the frequent contest mining using WEKA tools and Apriori algorithms to implement an association rule to find context pattern. Therefore, they argued that the generated context rules helped the developer to avoid the performance defect.

3.7 Analyzing Other Patterns

This section contained several studies in different fields of analyzing android source code. These fields included testing, energy, prioritizing and localization.

In testing field, there was one study of analyzing Android testing patterns. The study presented the mobile device platform was different than other platforms such as PC, screen size, touching the screen and sensor features in mobile platform require a user interaction. Also, user interaction quickly evaluated on library and APIs and fragmentation on mobile device platforms lead to new challenges in mobile app testing. However, manual testing on mobile was performed more than automated testing

due to a limitation of available testing tools. And, generated manual test case scenarios consumed a testing time in the development phase. However, recording testing scripts decreased the time.

So that, Linares-Vásquez et al. in [33] provided a GUI based model called MONKEYLAB which depended on analyzing to solve current challenges was in generating test scenarios. However, previous models cannot generate a test case for the unobserved event in an execution trace from an app source code. The aim of their study was to provide automating testing approach more powerful than manual testing. MONKEYLAB approach used to analyze to generate test cases, especially event scenarios for natural and unnatural user usage for apps. It followed four phases. The first phase was a record phase in order to record the execution event traces. Thus, the get event command was used to collect low-level event log even click type event or other events that happen during the execution an app. The second was mining phase is in order to analyze event traces. The vocabulary of GUI model was extracted from APK source code and event logs.

Furthermore, to analyze source code, the APK-analyzer was used to decompile source code using the dex2jar and Procyon tools [33]. Also, srcml tool used to convert the source code to XML and apktool to extract APK resources file. From XML and resources, APK-analyzer assigned GUI component and represented them as action and type tuples. For example, button represented as a click. Then, event logs analyzing was implemented by the data collector to collect a sequence of tokens from event logs which are represented GUI event. Each line in the event log was described by time stamp, input method, an action, property concern the actions and other property value.

Additionally, GUI component was extracted by translating low-level event logs into

the natural language description [33]. From GUI component, GUI tree, which included information about the component, its location and dimension was implemented by the Android view server was ruined in the device. Then, a component area was calculated by adding location to a dimension which decides the corresponding GUI component in order to avoid the dependency between events on a screen device. However, Using mining phase, GUI model was represented, including activities, component, action, and transition. Then, the third phase was generate phase in order to generate scenarios using dynamic and static analysis and the GUI model data which was extracted from mining phase used to build a statistical language model. And the fourth was validated phase to validate the generating scenarios through interaction with a real device.

Apart from that, in the energy consumption field, there was one study of analyzing Android energy patterns. The study showed that the increasingly demanding on mobile apps replace others platforms, such as desktop and web [34]. Through mobile apps, users satisfied whole needs by offering the most features. Therefore, mobile resources differed in different mobile versions and a user's usage. The mobile devices seemed a huge energy consumption. Conversely, a battery lifetime in mobile devices was limited. That leaded a user to rapid recharging device battery. Avoiding frequent discharge or using less energy consumption app might solve frequent recharging problems, but most benefits app were using a huge power for example video. Additionally, a high power consumption was caused by API misapplication or programming errors.

So that, Linares-Vásquez et al. in [34] presented a qualitative and quantitative study to show the causes of a high energy consumption on Android platform. Their study depended on data mining the API call method and usage patterns. When developers

used a suitable API and usage patterns, it helped to reduce a high energy consumption. The study was implemented in 55 Android 4.2 apps using Nexus 4 LG phone. Also in their study, the main dependent variable was energy consumption. In order to measure energy consumption, a set of scenarios was defined for 55 apps collected. The scenarios were recorded using recorded monkey tools. Execution of these scenarios was automated because of its long time.

Consequently, the estimation of energy consumption of collecting a data was divided into the three steps [34]. Firstly, the scenarios were monitored to collect execution trace from the activity manager profile. Secondly, the execution trace was analyzed to assign power measure to Android API methods. Then, API usage pattern was computed which its energy consumption computed by all invoked methods energy consumption. In their experiment, only the pattern of the length of 2-3 was selected. Thirdly, the signature of sequences for invoking a method was defined to match them in the source code. When their study was applied to the collected data, approximately 131 API method calls 18 patterns of length 2 and 8 patterns of length 3 were extracted. However, by referring to the source code, the API was concerned with the GUI, image manipulation, and database founded as the top energy greedy group.

Whereas, in the prioritizing field of Android apps, there was also one study of analyzing Android prioritizing patterns. The study showed that Android platform was produced by a huge various device manufacturer [35]. Android fragmentation drived main challenges in software engineering process. Moreover, mobile hardware such as screen size and resolution were taken into account by app developers. It's important for the developer to know which was a device model to gain more users. So, selecting a fit device model was a critical issue in app development.

So that, Lu et al. in [35] presented PRADA approach using mining the usage data onto apps in larger scale in order to prioritize the list of suitable device models for

an app. The data usage of the apps included the operational profiling that gives a quantification of the app using. PRADA utilized filtering techniques to predict how to use an app in different mobile models. PRADA also depended on the users of an app to decide the more priority device model. Then, it was evaluated by wandonujia data set. 200 apps from media and game categories was selected for evaluation. Then, the total browsing time that users interacted with an app was computed. For the dataset, around 14,7 thousand device model and 3 million users were recorded.

Again, there was one study in the field of analyzing Android localization patterns [34]. The localization was a process to adapt the software into a specific region, whereas the internationalization was a process to introduce the software that can be adapted to different languages and regions. Linares-Vásquez et al. in [34] showed a model to understand and collect information about internationalization, localization and translation through analyzing android repositories especially android SCM Logos. Through analyzing the source code to decide how localization occurs and who was responsible to do it and how the number of people needs to do. In their study, the analysis was done on Android R file by comparing the localization (es/values-xx/strings.xml) and internationalization file (es/values/strings.xml) through implementing a set of SQL queries. In their result showed that Android should make the localization process easily by obtaining specialized team to do it.

3.8 Highlight the Gap of knowledge

A little attention has been focused on activity lifecycle model development. However, there are no previous studies about analyzing Android source codes lifecycle usage patterns except one study that used analyzing to testing lifecycle conformance [3]. To fill this gap, our aim of this study is using a statistical analysis to analyze Android

source code by developing a tool. SAALC tool is able to analyze Android source code and explore the issues patterns related to lifecycle, especially callback methods and manage system resources to give community general statistics about lifecycle usage. Through using a methodology to analyze Android source code lifecycle patterns, this gives developers and researchers a vision about Android development practices by providing statistics in a lifecycle utilization from available open source code apps.

3.9 Conclusion

Modern research focuses on analyzing Android source code patterns. These researches provide new ways and patterns of different fields like design, privacy, security, quality, localization, prioritizing, energy and testing patterns. The patterns aids developers to produce more efficient, tolerance, high quality and reliable application. Analyzing lifecycle coding pattern has not used in modern study yet. It helps to understand the behaviors and experiences of developers to give novice and amateur developers guidelines for development correct activity which conform to lifecycle model.

The understanding activity model lifecycle is very difficult for the beginner developers. Actually, there are limited tools that used in lifecycle conformance. Moreover, as mentioned before, some studies show that the Android activity lifecycle model and documentation are informal, inconsistent and incorrect. However, these studies overcomes these problems by rebuilding a new model of activity lifecycle, but it still affects the developer behaviors.Therefore, analysis Android source codes and find patterns related to losing data and resource management was introduced in our study to understand the developers behaviors. Additionally, it gives new developers and researchers a view of current lifecycle utilization of development activities.

In the next chapter, the methodology of this thesis was presented. The methodology considered Android dataset collection and analysis using our implemented tool.

Chapter 4

Research Methodology

Analyzing a huge dataset of Android source code for exploration lifecycle usage patterns was not introduced before. To fill this gap, a large dataset was collected to explore these patterns. The patterns were considered as useful information and statistics which is extracted from a dataset. Our research contributes to use analyzing of Android source codes in order to provide a general statistics about lifecycle patterns for the community. This chapter presented our research methodology. Section 4.1 showed a flow of the research method. In section 4.2, it showed our data collection. And, section 4.3 showed our data analysis approach.

4.1 The Research Methodology Flow

The main aim of this study is to explore how real Android developers develop their apps in terms of lifecycle callback methods. To address this need, a quantitative methodology was applied to explore patterns using analyzing of Android source codes. Analyzing was implemented on a dataset of Android activities to extract information about the usage of lifecycle's callback methods, managing (acquiring and releasing) system's resources and the nature of code inside callback methods.

In this section, our research methodology flow was presented. It was structured depending on two previous studies. The first study is [4] which was guided us in how selecting a dataset of Android activities, analyzing its source codes and displaying results. Also, the second is [3] which was guided us in introducing a static analysis for Android's activities.

This research methodology was implemented in five steps. Figure 4.1 shows the phases of our research methodology using a flow diagram. These phases are:

1. Selection phase: F-Driod repository was selected to obtain Android apps.
2. Pre-processing phase: all activities were extracted from each apps.
3. Transformation phase: each activity was transformed into Abstract Syntax Tree (AST) object model. AST object was used to get information about each activity's source code such as packages, fields and methods declarations.
4. Analysis phase: an analysis tool called a Statistical Analysis of Android Lifecycle (SAALC) was developed to analyze Android activities.
5. Extraction phase: some patterns were extracted after the analysis phase. These patterns concern about lifecycle usages.
6. Representation phase: statistics were collected from the patterns. Then, reports and indications were generated which offered a feedback to the community about Android lifecycle usages. Further, effective visualizations were used to draw results of our analysis using graphs such as Column, Heat and Column map, Pareto and Bubble Chart.

FIGURE 4.1: Methodology flow diagram

4.2 Data Collection

Implementation the analyzing for Android source code requires a dataset of Android apps. This section described how collecting our dataset.

During the Selection phase, a repository called a Free and Open Source Software applications for the Android (F-Droid) was selected [36]. Then, a dataset of Android's apps was collected from F-Droid. The apps collection methodology was followed as proposed in Figure 4.2. It includes five steps:

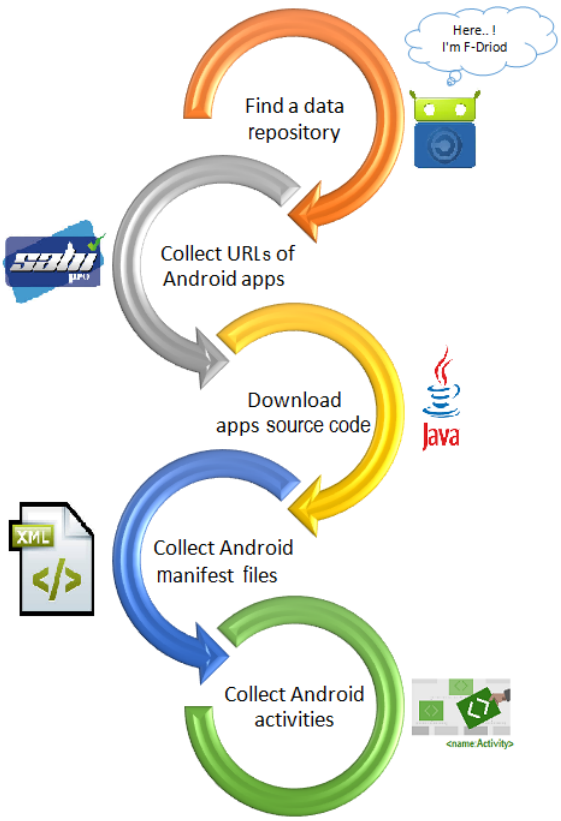


FIGURE 4.2: Data collection methodology

1. Find a data repository: after large sets of related work were studied, this helped us finding the Android apps repository. The repository F-Droid was selected.

   F-Droid is a popular platform and an online software repository which contains open source code for Android apps [36]. It is invoked through a link: "https://fossdroid.com/". Most apps in F-Droid is also available on Google play [36]. F-Droid was selected because it provides a categorization of the Android apps. This categorization helped in analyzing Android apps according to the app categories. At Dec 8, 2016, F-Droid contained 2001 apps (1420 also on Google Play) organized in 17 categories [37]. The distribution of a Number of Apps (#App) is showed in Table 4.1.

2. Collect URLs for Android apps: URLs for the apps which was stored in F-Droid

and hosted on GitHub was collected. The URLs collection process was automatically done. A script was written to collect URLs for each category and save them in a file. Then, Sahi tool was used to perform the automated app's URLs collection [38]. However, some problems were faced during the downloading apps using Sahi, because of the size of apps source code is very large and needs lot time. In order to overcome that, the downloading app process was done in a different way as shown in the next step.

3. Download apps source code: a piece of JAVA code was written to download app's source code using the app's URL. The apps were downloaded from its the individual pages. For every app's URL, JAVA code was connected to it, downloaded a zip file of source code and decompressed it. In total, 842 apps were downloaded in our dataset from 17 categories. The distribution of a Number of Downloaded Apps (#App Downloaded) shown in Table 4.1.

4. Define Android's manifest file: for every 842 apps, a piece of JAVA code was used to search for the manifest files, parse it using XML parser and obtain the names of all activities inside it using "<activity android:name>" tag.

5. Collect Android's activities: activities names were used to do an automated search also using JAVA code of the activity files. The activity files were copied into our dataset library. However, the activities file names were changed to avoid the conflict with other activities that have the same name. In total, 5577 activities were collected and organized into 17 categories. The distribution of a Number of Activities (#Activity) shown in Table 4.1.

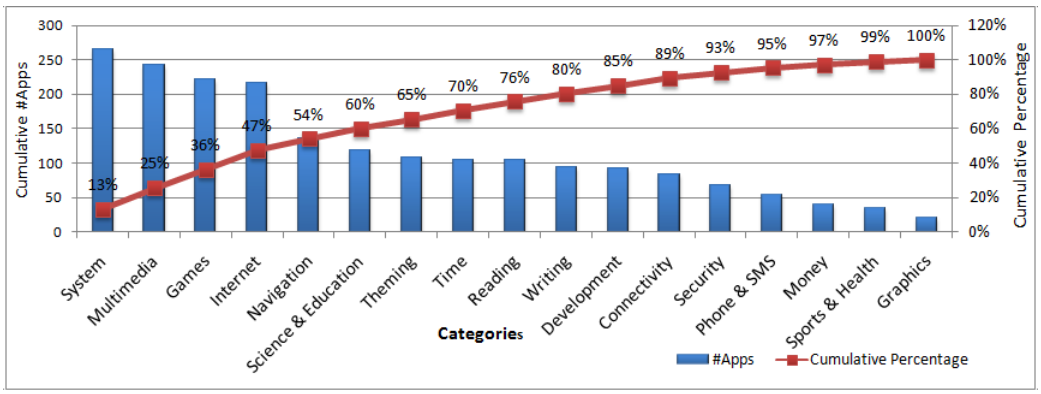
Consequently, the dataset of our research includes 5577 activities extracted from 842 Android apps, which organized into 17 categories as shown in Table 4.1. Table 4.1 shows that *System* category has the largest #APP then *Multimedia* [36]. Moreover, *Internet* category has the largest #App Downloaded then *Multimedia*. And, the largest

#Activity was found in categories *Internet* then *Science & Education*.

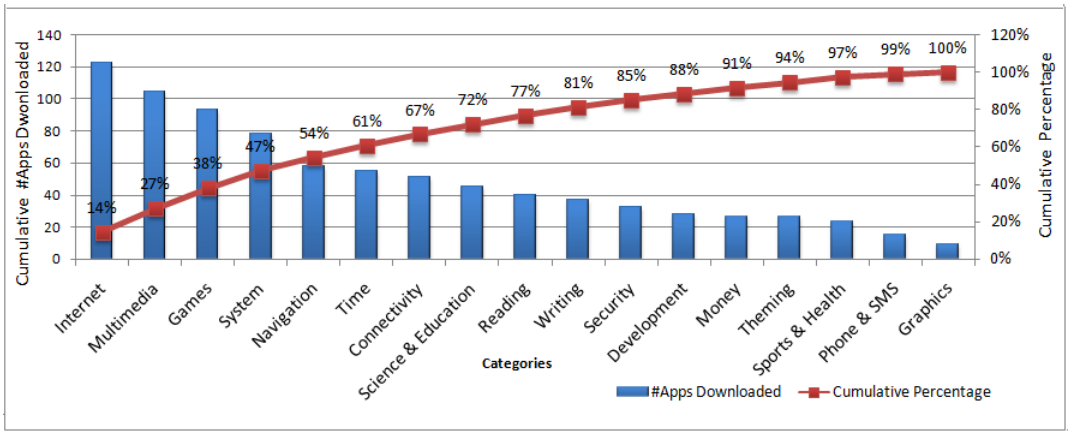
TABLE 4.1: Distribution of the dataset over the app categories

Category	#APP	#App Downloaded	#Activity
System	265	78	325
Multimedia	242	104	532
Games	221	93	422
Internet	217	122	952
Navigation	135	58	370
Science & Education	118	45	624
Theming	108	26	94
Reading	104	40	347
Time	104	55	328
Writing	94	37	242
Development	92	28	182
Connectivity	84	51	236
Security	68	32	215
Phone & SMS	53	15	159
Money	40	26	266
Sports & Health	35	23	247
Graphics	21	9	36
Total	2001	842	5577

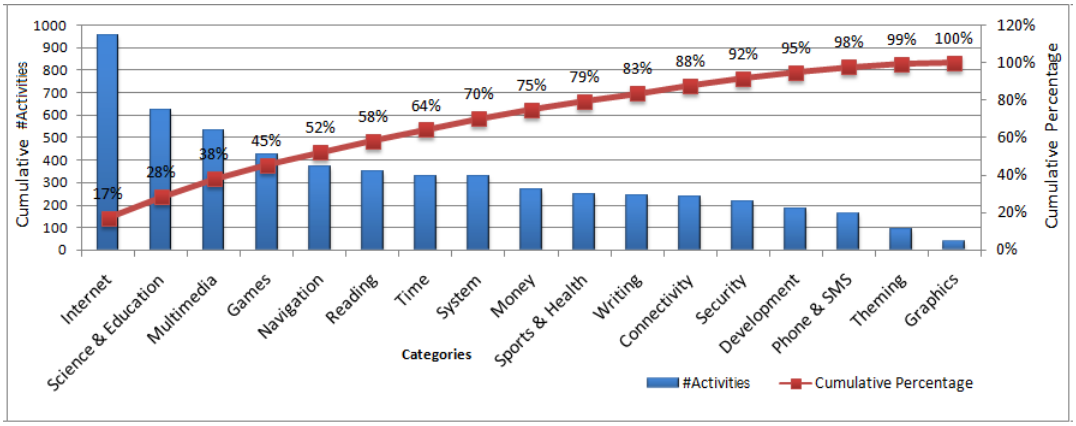
Each sub figure 4.3a,4.3b and 4.3c show a Pareto Chart. The Pareto Chart gives our dataset a graphical distribution using a combination of a line and bar graphs. The sub figures contain a bar chart which represents the cumulative totals of #App, #App Downloaded and #Activities across each category, while the line graph shows the cumulative percentages for them. The Pareto chart reveals that 45% of activities in the dataset belong to 23% of the categories inside the first four categories which are *Internet, Science & Education, Multimedia,and Games*.



(A)



(B)



(C)

FIGURE 4.3: Distribution of the dataset over 17 categories in F-Droid: (A) Number of F-Driod apps(#App). (B) Number of downloaded apps(#App Downloaded). (C) Number of collected activities (#Activities)

4.3 Data Analysis

Analyzing Android source code is very important to provide feedback and indications for the community about actvity's lifecycle usage. The strong issue here that this approach of analyzing has not applied before for Android. So, our analyzing methodology was applied depending on others source codes analyzing platform such as JAVA which was followed in this research as showed by Lamba et al. study[4].

In this study, developers and interested researchers were helped through showing the usage of the mobile activity lifecycle by automating the process of analyzing source code against the lifecycle model and documentation. The goal of analyzing methodology was to obtain hidden patterns from Android apps dataset. Then, statistics and reports were generated from these patterns to return a basic information for the community about callback methods and system's resources. To do the analyzing process, SAALC tool was developed. This section showed a detail about SAALC's architecture, implementation and testing.

4.3.1 SAALC Architecture and Implementation

SAALC is able to read a huge dataset of Android activities and use the lifecycle's model rules to analyze it. SAALC is the first tool which implemented the analyzing a huge dataset of Android activities. Figure 4.4 depicts our proposed analyzing approach using a structure diagram of SAALC tool. It shows a block diagram of the main components of SAALC as well as the approach of the analysis process.

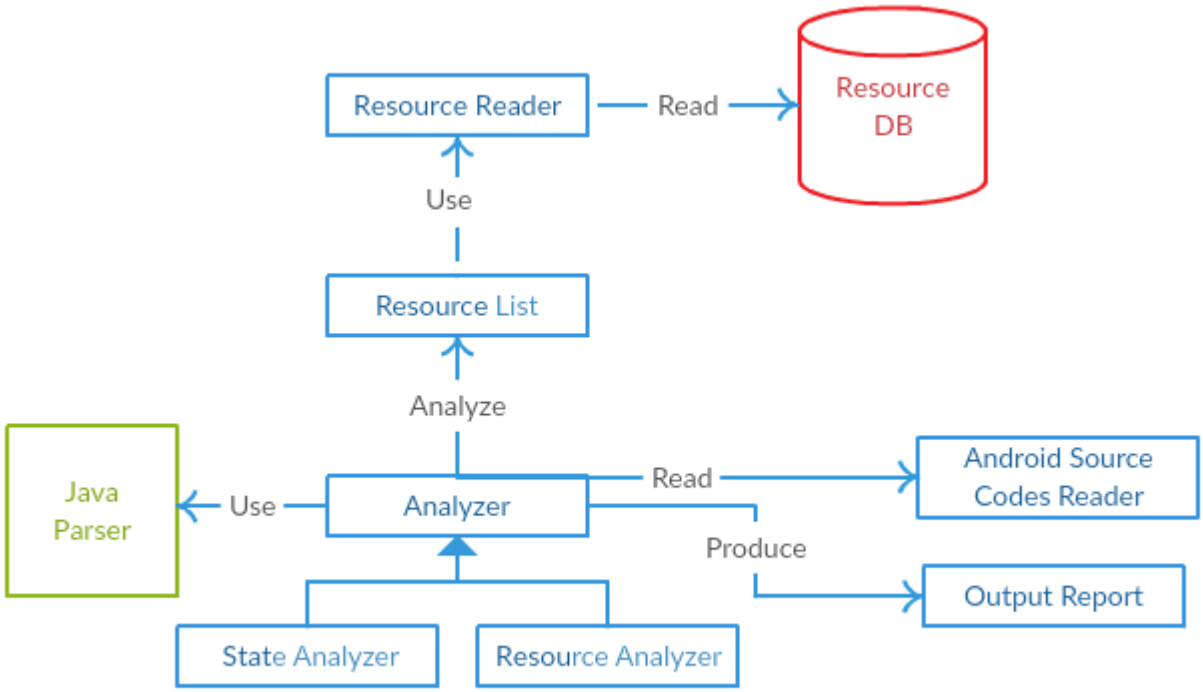


FIGURE 4.4: Structure diagram of SAALC

The block diagram of SAALC includes these component:

- Java Parser component: it is an open source and free parser available on GitHub, it uses to parse and convert the Android source code into AST [39]. The AST object contains a list of import packages, methods, field declarations for each class of a source code. These declarations were used in our analysis. Using an import package declaration, the names of packages that declared in the class can be decided. Additionally, using a field declaration, the field names and its type can be decided. And, using a method declaration, method names and its body contents can be decided.
- Android Source Codes Reader component: it reads a dataset of Android's activates.

- Output Report component: it produces output reports in Comma Separated Values (CSV) file format.
- Resource List component: it produces a list of resources.
- Resource DB component: it stores resources information in an XML file.
- Analyzer component: it's the main component in the diagram. Analyzer component applies two handling algorithms. So, it inherits in two types:
  - State Analyzer component: it inspects dataset source code to collect statistics about callbacks methods and the natures of code inside them.
  - Resource Analyzer component: it inspects dataset source code to collect statistics information about managing system's resources.

In general, the steps are described in our approach as follows. First, SAALC reads the system resources information from the repository using Read Resource component, then produce a list of resources using Resource List component. Secondly, JAVA Parser component parses Android source code and produces AST object model. Then, using the resulting object model produces by Java Parser, the tool applies two types of algorithms. The first algorithm called State Analyzer, which is responsible to collect information about each callback methods such as the count of each callback method and the nature of code inside them. The second algorithm is called Resource Analyzer that inspects the source code against system resources list. The tool produces results report in CSV file format using Output Report component.

The system's resources were stored in Resource DB component in the XML file repository. XML file dynamically increases the tool's ability to insert new information resource. The information for 9 system resources was stored such as *Camera, USB, Sensor, Network, Input, GPS, Database, Bluetooth, and Audio*. This information includes the resource name, package names, the name of acquiring and releasing methods and

the name of callback methods which used to acquire and release the resource. All above information was taken from the official Android site [17]. Moreover, these information were used to extract patterns from the dataset. At Appendix A.1, Table A.1 shows the repository information for the nine system's resources.

In Analyzer component, the Analyzer algorithm was built in JAVA language. This algorithm is able to reach all statements in any method's body. Also, it is able to analyze the common coding patterns and styles. Below are the common coding style is used by developers [3]:

- Developer calls the acquired or released method directly inside a callback method block as shown in Figure 4.5a.
- Developer calls another method or nested methods inside a callback method which in turn calls the acquired or release method as shown in Figure 4.5b.
- Developer call the acquired or release method inside if, while, for, switch, try catch, threads or object block statements which are inside the callback method or other nested methods as shown in Figure 4.5c.

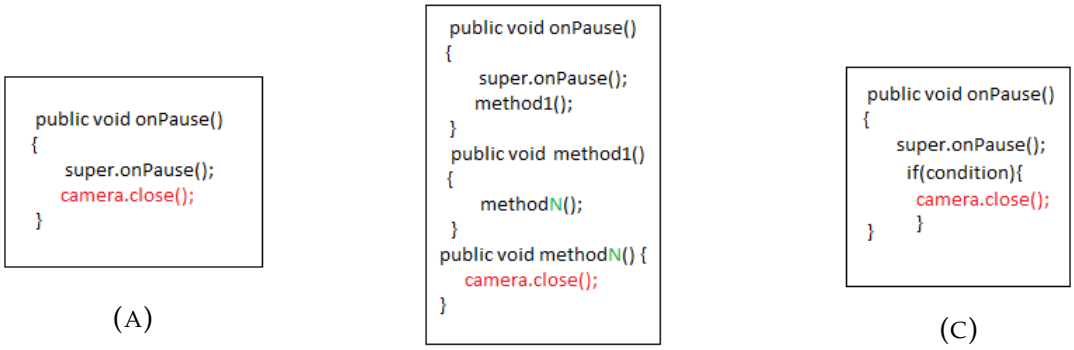


FIGURE 4.5: Common coding styles

Sometimes, developers did not acquire or release system's resources inside callback methods. Instead, they use different methods which do implement other kinds of

events. Android source code was analyzed against these events. These events included methods which were overridden in the activity source code and did not a callback method. These event methods were referred in the result section using "OTHER" keyword.

However, the code analyzing process was divided into two parts:

4.3.1.1 State Analyzer Algorithm

The proposed algorithm for State Analyzer can be described in pseudo-code as shown in as follows:

- **Algorithm Input:** List of all activities source code in our dataset.
- **Algorithm Output:** A report result in CSV (occurrences of callback methods).

Algorithm basic steps:

1. Load a list of activities source code.
2. For each activity in the list of activities, parse and traverse the activity source code into AST. Then, analyze the source code to find and count the occurrences of the callback methods which were used.
3. If onPause() ,onDestroy() or onStop() callback methods were founded in the source code, then analyze the nature of code inside each of them related to releasing, database or threading actions.

4.3.1.2 Resource Analyzer Algorithm

The proposed algorithm for Resource Analyzer can be described in pseudo-code as shown in as follows:

- **Algorithm Input:** List of all activities source code, List of system's resources information.
- **Algorithm Output:** A report result in CSV(occurrences of system's resource in the callback methods).

Algorithm basic steps:

1. Load a list of activities source code, and list of resource information. This includes all resource names, package names, names of acquiring and releasing methods.
2. For each activity in the list of activities, parse and traverse the activity source code into AST.
3. For each system resources such as *Camera, GPS* etc, analyze the source code of each activity to find in any methods (callback or OTHER methods) where this resource has been acquired and released.

For example, when Resource Analyzer component tries to check *Camera* resource occurrence, it searched for all *Camera* resource package name such as "Android.hardware.camera" as shown in Table A.1. If the analyzer finds it, then it searches for the names of variables that developer initialized for *Camera* resource in the fields declaration list. So, if the analyzer finds the field declaration as "Camera camera1", then the name of *Camera* resource that used by the developer is "camera1". The analyzer then checks all callback or "OTHER" methods inside the activity, then return all statements which declared in each of them. If the analyzer finds any statement which contains *Camera* resource variable name and the method that used for acquiring *Camera* resource, then it increases the count of acquired *Camera* resource for this method. For instance, if analyzer finds "camera.open()" statement inside the onCreate() callback method, that's mean developer acquired *Camera* resource inside the onCreate(). Whereas, if

the analyzer finds any statement that contains *Camera* resource variable name and the method that used for releasing *Camera* resource, then the analyzer increases the count of released *Camera* resource for this methods. For instance, if the analyzer finds "camera.close()" statement inside the onPause() callback method, that's mean developer released *Camera* resource inside the onpause().

4.3.2 SAALC Implementation

SAALC was built in JAVA programming languages. At Appendix A.3 Figure A.1 shows a deep descriptions for associations between classes inside SAALC using a class diagram implementation.

4.3.3 SAALC Testing

SAALC tool deals with a large dataset of Android's activities and that's made our analysis very hard. To implement SAALC testing, firstly it used for analyzing a sample of 10 activities. The testing was implemented through log statements such as the print statement and debugging steps of the codes. When the correct result was ensured from this sample, then the number of activities was increased gradually until reach to 50 activities. Finally, the analyzing process was applied to 5577 activities in our dataset.

In the next Chapter, the extracted patterns and statistics were generated after analyzing the collected dataset using SAALC tool.

Chapter 5

Results

In this section, the result of our study was shown after the analysis process was applied to 5577 Android's activities. The result was divided into three parts as shown in the following subsections 5.1,5.2 and 5.3. The first section showed the results of the callback methods usage, the second for the resource usage and the third for the nature of code inside some of the most influences callback methods such as onPause(), onStop() and onDestroy().

5.1 Usage of Callback Methods

In this section, the percentages of occurrence of the activity callback methods over the data set were founded. Moreover, it showed the percentages of occurrences for the callback methods over the app categories.

5.1.1 Percentages of the usage of callback methods over the dataset

In the first part, State Analyzer algorithm was run to count the total of occurrences for each callback method for the 5577 activities. The result set was presented in Table 5.1.

TABLE 5.1: Distribution of callback methods over the dataset

Callback method	#Activities	%Activities
onCreate()	5115	92%
onResume()	1299	23%
onPause()	888	16%
onDestroy()	780	14%
onStart()	346	6%
onStop()	341	6%
onRestart()	31	1%

Table 5.1 shows the number of each callback methods was founded in the dataset. Also, it shows the percentages of occurrences for each callback methods count over 5577 activities. On the other hands, these results was represented in Figure 5.1 which shows a Bubble Chart. The Bubble Chart considers three dimensions: (i)the horizontal axis represents the callback methods names; (ii)the vertical axis represents the counts of occurrence of each callback method; (iii) the bubble size indicates the third dimension which represents the cumulative percentage of the callback methods. The Bubble Chart also shows that the most occurrences callback method is 92% for onCreate() followed by 23% for onResume() then 16 for onPause() , 14% for onDestroy(), 6%s for onStart() and onStop() and the last is 1% onRstart().

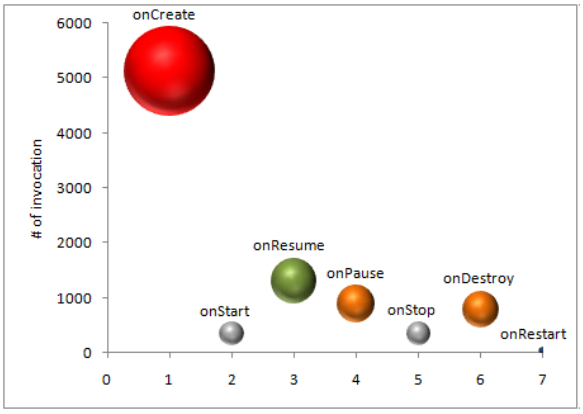


FIGURE 5.1: Buble Chart: Distribution of callback methods over the dataset

5.1.2 Percentages of the usage of callback methods over the app categories

Our analysis of callback methods was also produced in relation to the categories of the Android apps. The counts and percentages of callback methods were shown according to the F-Driod categories distributions in Table 5.2.

TABLE 5.2: Distribution of callback methods over the apps categories

Category	onCreate()	onStart()	onResume()	onPause()	onStop()	onDestroy()	onRestart()
Connectivity	(224) 4%	(28) 1%	(59) 1%	(46) 1%	(21) 0%	(77) 1%	(4) 0%
Development	(167) 3%	(14) 0%	(38) 1%	(22) 0%	(9) 0%	(18) 0%	(0) 0%
Games	(392) 7%	(22) 0%	(107) 2%	(93) 2%	(21) 0%	(49) 1%	(1) 0%
Graphics	(35) 1%	(0) 0%	(11) 0%	(3) 0%	(3) 0%	(7) 0%	(2) 0%
Internet	(888) 16%	(77) 1%	(261) 5%	(181) 3%	(80) 1%	(138) 2%	(4) 0%
Money	(227) 4%	(8) 0%	(79) 1%	(44) 1%	(10) 0%	(44) 1%	(0) 0%
Multimedia	(514) 9%	(49) 1%	(133) 2%	(108) 2%	(35) 1%	(100) 2%	(8) 0%
Navigation	(309) 6%	(26) 0%	(95) 2%	(68) 1%	(24) 0%	(35) 1%	(1) 0%
Phone & SMS	(150) 3%	(10) 0%	(46) 1%	(26) 0%	(6) 0%	(19) 0%	(0) 0%
Reading	(331) 6%	(16) 0%	(55) 1%	(48) 1%	(15) 0%	(46) 1%	(2) 0%
Science & Education	(550) 10%	(11) 0%	(115) 2%	(46) 1%	(22) 0%	(35) 1%	(0) 0%
Security	(196) 4%	(10) 0%	(50) 1%	(38) 1%	(11) 0%	(26) 0%	(2) 0%
Sports & Health	(215) 4%	(12) 0%	(28) 1%	(19) 0%	(13) 0%	(30) 1%	(0) 0%
System	(295) 5%	(26) 0%	(74) 1%	(42) 1%	(28) 1%	(59) 1%	(4) 0%
Theming	(70) 1%	(4) 0%	(14) 0%	(4) 0%	(3) 0%	(14) 0%	(0) 0%
Time	(317) 6%	(20) 0%	(68) 1%	(51) 1%	(32) 1%	(54) 1%	(2) 0%
Writing	(235) 4%	(13) 0%	(66) 1%	(49) 1%	(8) 0%	(29) 1%	(1) 0%

Figure 5.2 displays a Heat-Map according to Table 5.2. Heat-Map which is a graphical representation of the callback methods frequency data across categories; where the individual values contained in the matrix are represented as colors. The larger values are represented by relatively darker colors in comparison to smaller values which are represented by lighter colors. Further, the three dimensional data displayed by the Heat-Map in terms of activity frequencies across all the categories. The formula for

computing the individual values contained in the matrix represented as colors are based on the percentages of occurrence for the callback method in a category.

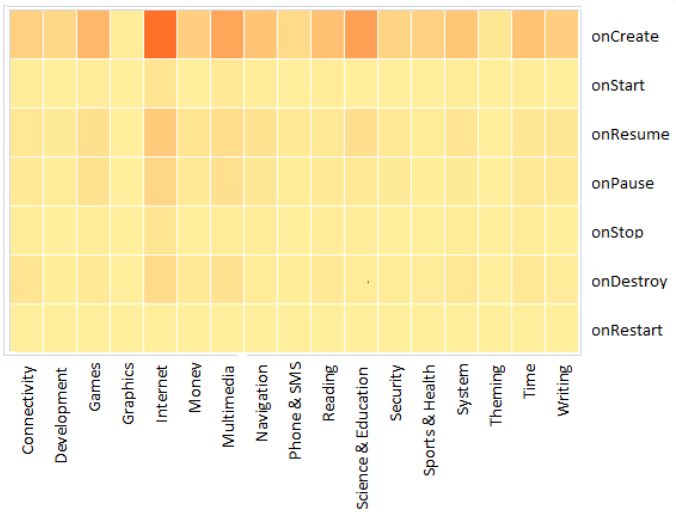


FIGURE 5.2: Heat-Map: Distribution of callback methods over the apps categories

The map shows that:

- The highest value is (888) 16% for the onCreate() callback method and category *Internet* followed by a value (550) 10% for the onCreate() and category *Science & Education* and then value (514) 9% for the onCreate() and category *Multimedia*.
- The lowest values for the onRestart() callback method across all categories.
- The most frequent use is for *Internet* category at all callback methods followed by *Multimedia* then *Science & Education*.

5.2 Usage of System's Resources

The second part of the analyzing process was finding the usage of the system's resources. The usage of system's resources was considered the occurrence of system's

resources in the dataset. Moreover, managing (acquiring and releasing) system's resources. In this section, the percentages of the system's resource usage were shown.

5.2.1 Occurrence of the system's resources

In this section, the focus was on finding the occurrence (persistence) of the system's resources into the dataset. Then, the result was also shown the persistence of the system's resources in relation to the app categories.

5.2.1.1 Percentages of system's resources occurrence over the dataset

Resource Analyzer was run to find the occurrences of each system's resources over 5577 activities. The result was shown in Table 5.3.

TABLE 5.3: Distribution of system's resources over the dataset

Resource	#occurrence
Database	(52) 0.93%
Sensor	(26) 0.46%
Camera	(21) 0.37%
USB	(19) 0.34%
Input	(18) 0.32%
Audio	(15) 0.26%
Network	(13) 0.23%
GPS	(9) 0.16%
Bluetooth	(5) 0.08%
Total	(178) 3%

Table 5.3 shows the Number and Percentages of Occurrences (#occurrence) for each system's resources founds in the dataset. The analyzer was run over nine resources which are *Camera, Audio, Bluetooth, Database, GPS, Input, Network, Sensor, and USB*. The total number of the resource is (178) 3% over 5577 activities. Figure 5.3 shows Column Chart which represents #occurrence of each system's resource.

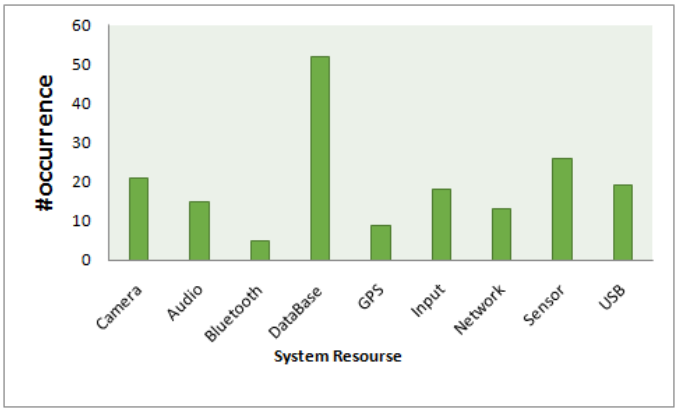


FIGURE 5.3: Column Chart: Distribution of system's resources over the dataset

The Column Chart shows that:

- The most popular system's resource occurrence used by developers is *Database* equals to (52) 0.93% followed by *Sensors* equals to (26) 0.46% and *Camera* equals to (21) 0.37%.
- The lowest popular system's resource occurrence used by developers is *Bluetooth* equals to (5) 0.08%.

5.2.1.2 Percentages of system's resources occurrence over the apps categories

Further, in this section, the analysis of system's resource occurrence was presented in relation to the categories of the Android apps. The counts and percentages of the system's resources were shown according to the app categories distribution in Table 5.4.

TABLE 5.4: Distribution of system's resources over the apps categories

Category	Database	Sensor	Camera	USB	Input	Audio	Network	GPS	Bluetooth
Connectivity	(0) 0%	(1) 0.02%	(1) 0.02%	(1) 0.02%	(0) 0%	(2) 0.04%	(5) 0.09%	(0) 0%	(2) 0.04%
Development	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Games	(4) 0.07%	(4) 0.07%	(0) 0%	(0) 0%	(0) 0%	(3) 0.05%	(0) 0%	(0) 0%	(1) 0.02%
Graphics	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Internet	(3) 0.05%	(1) 0.02%	(2) 0.04%	(1) 0.02%	(5) 0.09%	(1) 0.02%	(2) 0.04%	(0) 0%	(0) 0%
Money	(8) 0.14%	(0) 0%	(9) 0.16%	(1) 0.02%	(2) 0.04%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Multimedia	(17) 0.30%	(3) 0.05%	(4) 0.07%	(14) 0.25%	(2) 0.04%	(6) 0.10%	(0) 0%	(0) 0%	(0) 0%
Navigation	(1) 0.02%	(11) 0.20%	(0) 0%	(0) 0%	(1) 0.02%	(0) 0%	(3) 0.05%	(8) 0.14%	(0) 0%
Phone & SMS	(0) 0%%	(2) 0.04%	(2) 0.04%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Reading	(1) 0.02%	(2) 0.04%	(0) 0%	(0) 0%	(2) 0.04%	(0) 0%	(2) 0.04%	(0) 0%	(0) 0%
Science & Education	(8) 0.14%	(1) 0.02%	(1) 0.02%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(1) 0.02%
Security	(4) 0.07%	(0) 0%	(0) 0%	(0) 0%	(3) 0.05%	(0) 0%	(1) 0.02%	(0) 0%	(0) 0%
Sports & Health	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(2) 0.04%	(0) 0%	(1) 0.02%	(1) 0.02%
System	(2) 0.04%	(0) 0%	(2) 0.04%	(2) 0.04%	(2) 0.04%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Theming	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Time	(0) 0%	(1) 0.02%	(0) 0%	(0) 0%	(0) 0%	(1) 0.02%	(0) 0%	(0) 0%	(0) 0%
Writing	(4) 0.07%	(0) 0%	(0) 0%	(0) 0%	(1) 0.02%	(0) 0%	(0) 0%	(0) 0%	(0) 0%

Table 5.4 was represented in Figure 5.4. Figure 5.4 displays a Column-Map which is a graphical representation of the system's resources frequency data across categories where the individual values contained in the matrix are represented as column. The larger values are represented by relatively high columns in comparison to smaller values which are represented by lighter columns. The three dimensional data displayed by the Column-Map in terms of activity frequencies across all the categories. The formula for computing the individual values contained in the matrix represented as columns is the percentages of occurrence for the system's resources in categories.

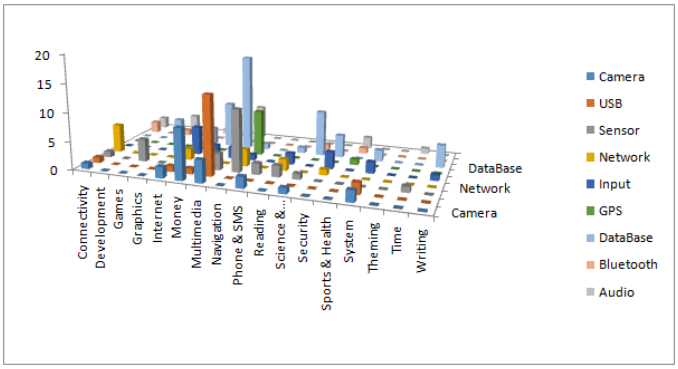


FIGURE 5.4: Column-Map: Distribution of system's resources over the apps categories

The Column-Map shows that:

- The highest value is (17) 0.30% for *Database* system resource and category *Multimedia* followed by a value (14) 0.25% for *USB* and category *Multimedia* then (11) 0.20% for *Sensor* and (8) 0.14% for GPS at category *Navigation*.
- The lowest values for *Bluetooth* resource across all categories.
- The most frequent use for *Database* resources in *Multimedia* category.

5.2.2 Managing system's resources

Managing system's resources give researchers indication about how developers deal with acquiring and releasing system's resources during the activity's lifecycle. In this section, the other information was extracted in the analysis was to find the most popular acquired and released recourse in the callback methods.

5.2.2.1 Percentages of acquiring system's resources

The Resource Analyzer was run In order to find the most popular acquired resources. The results of this analysis were shown in Table 5.5.

TABLE 5.5: Distribution of acquired system's resources

Category	onCreate()	onStart()	onResume()	onPause()	onStop()	onDestroy()	onRestart()	OTHER
DataBase	(44) 85%	(0) 0%	(4) 8%	(4) 8%	(0) 0%	(8) 15%	(0) 0%	(35) 67%
Sensor	(26) 100%	(0) 0%	(2) 8%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(1) 4%
Camera	(4) 19%	(1) 5%	(7) 33%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(17) 81%
USB	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Input	(6) 33%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Audio	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Network	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
GPS	(1) 11%	(0) 0%	(1) 11%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(1) 11%
Bluetooth	(2) 40%	(1) 20%	(1) 20%	(0) 0%	(0) 0%	(1) 20%	(0) 0%	(2) 40%

Table 5.5 shows the most popular callback methods that used by developers to acquire each resource. The result set was also represented in Figure 5.5.

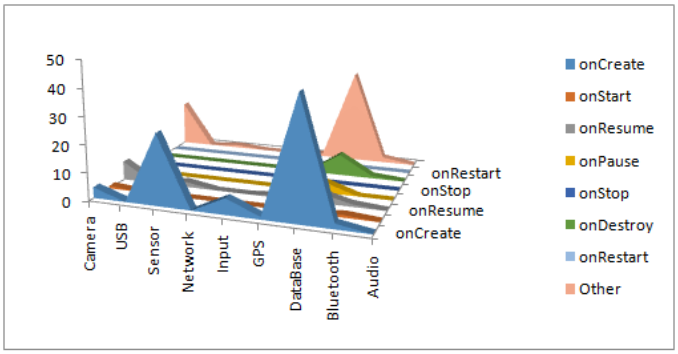


FIGURE 5.5: Column Chart: Distribution of acquired system's resources

The results show that:

- *Database* was acquired mostly on the onCreate() callback method with the percentage of occurrence equal to 85% over 52 activities which used *Database* resource. Also, 8% used the onPause(), 8% the onResume(), and 15% used the onDestroy(). Whereas, around 67% of activities acquired *Database* on OTHER methods.

- *Sensor* was acquired on the onCreate() method with the percentage of occurrence equal to 100% over 26 activities which used *Sensor* resource. Also, 8% used by the onResume(). Whereas, around 4% of activities acquired *Sensor* on OTHER methods.
- *Camera* was acquired mostly on OTHER methods with the percentage of occurrence equal to 81% over 29 activities which used *Camera* resource. Also, 33% used the onResume(), 19% used the onCreate(), and 5% used the onStart().
- *Input* was acquired mostly on the onCreate() with the percentage of occurrence equal to 33% over 18 activities which used *Input* resource.
- *GPS* used 11%s of activities over 9 activities which acquired *GPS* resource in the onCreate(), onResume() and OTHER methods.
- *Bluetooth* was acquired mostly on the onCreate() and OTHER methods callback methods with percentage of occurrence equal to 40% over 5 activities which used *Bluetooth* resource. Also, 20%s used the onStart(), onResume(), and onDestroy().
- *USB*, *Audio*, and *Network* resources had nothing of the percentages of acquired.

5.2.2.2 Percentages of releasing system's resources

The same as 5.2.2.1 sub section, Resource Analyzer was run In order to find the most popular released resources. The results of this analysis were shown in Table 5.6.

TABLE 5.6: Distribution of released system's resources

Resource	onCreate()	onStart()	onResume()	onPause()	onStop()	onDestroy()	onRestart()	OTHER
Database	(0) 0%	(0) 0%	(0) 0%	(2) 4%	(0) 0%	(8) 15%	(0) 0%	(0) 0%
Sensor	(1) 4%	(0) 0%	(0) 0%	(1) 4%	(0) 0%	(0) 0%	(0) 0%	(1) 4%
Camera	(1) 5%	(0) 0%	(2) 10%	(12) 57%	(1) 5%	(3) 14%	(0) 0%	(12) 57%
USB	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Input	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Audio	(7) 47%	(0) 0%	(0) 0%	(3) 20%	(0) 0%	(1) 7%	(0) 0%	(2) 13%
Network	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
GPS	(1) 11%	(0) 0%	(0) 0%	(1) 11%	(0) 0%	(1) 11%	(0) 0%	(1) 11%
Bluetooth	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%

Table 5.6 shows the results of the most popular callback method used to release resources. It was represented in Figure 5.6.

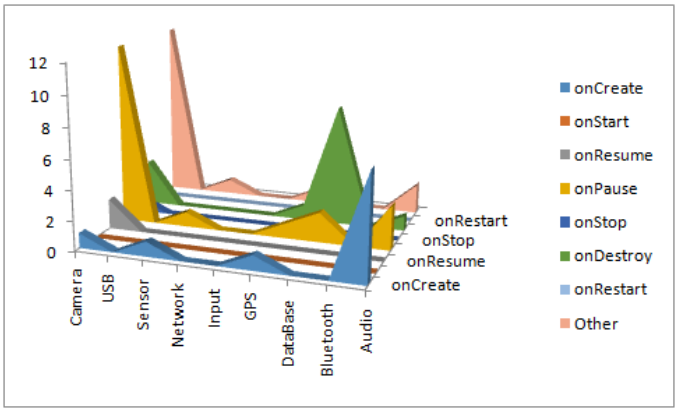


FIGURE 5.6: Column Chart: Distribution of released system's resources

The results show that:

- *Database* was released mostly on the onDestroy() callback method with the percentage of occurrence equal to 15% over 52 activities which used *Database* resource. Also, 4% used the onPause().
- *Sensor* used 4%s of activities over 26 activities which released the onCreate(), onPause(), and OTHER methods.

- *Camera* was released mostly on the onPause() method with the percentage of occurrence equal to 57% over 29 activities which used *Camera* resource. Also, 5% used the onCreate(), 10% used the onResume(), 5% used the onStart(),14% used the onDestroy(). whereas, around 57% of activities released *Camera* on OTHER methods.
- *Audio* was released mostly on the onCreate() with the percentage of occurrence equal to 47% over 18 activities which used *Audio* resource. Also, 20% used the onPause() and 7% used the onDestroy(). Whereas, around 13% of activities released *Audio* on OTHER methods.
- *GPS* used 11%s of activities over 9 activities which released *GPS* resource in the onCreate(), onPause(), onDestroy() and OTHER methods.
- *USB*, *Input*, *Network* and *Bluetooth* resources had nothing of the percentages of released.

5.2.2.3 Correctly/Wrongly acquired and released system's resources over the dataset

In order to obtain more supportive results, the correctly acquired and released percentages were decided depending on the Android documentation information at Table A.1. For each system's resources, the callback methods which are responsible to acquire and release resource were decided. Then, the value of percentages of these callback methods were decided as the correctly percentages of acquired and released the resource. On the other hands, the average of wrongly acquired and released percentages was computed by finding the averages of the callback methods percentages that registered acquired and released percentages and did not have a responsibility to do that. Table 5.7 shows these comparisons of correctly/wrongly acquired and released percentages of the system's resources.

TABLE 5.7: Distribution of correctly/wrongly acquired and released system's resources

Resource	Correctly acquired	Average of wrongly acquired	Correctly released	Average of wrongly released
Database	85% on onCreate()	25%	4% on onPause()	15%
Sensor	8% on onResume()	52%	4% on onPause()	4%
Camera	33% on onResume()	35%	57% on onPause()	18%
USB	0% on onResume()	0%	0% on onPause()	0%
Input	33% on onCreate()	0%	0% on onPause()	0%
Audio	0% on onCreate()	0%	20% on onPause()	22%
Network	0% on onCreate()	0%	0% on onPause()	0%
GPS	11% on onCreate()	11%	11% on onPause()	11%
Bluetooth	40% on onCreate()	25%	0% on onPause()	0%
Total Average	23%	16%	11%	8%

The result set of correctly/wrongly acquired percentages was represented in Figure 5.7. Whereas, The result set of correctly and wrongly released percentages was represented in Figure 5.8.

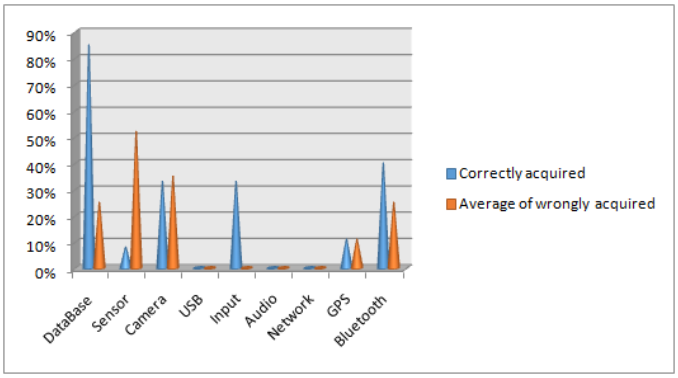


FIGURE 5.7: Column Chart: Distribution of correctly/wrongly acquired system's resources

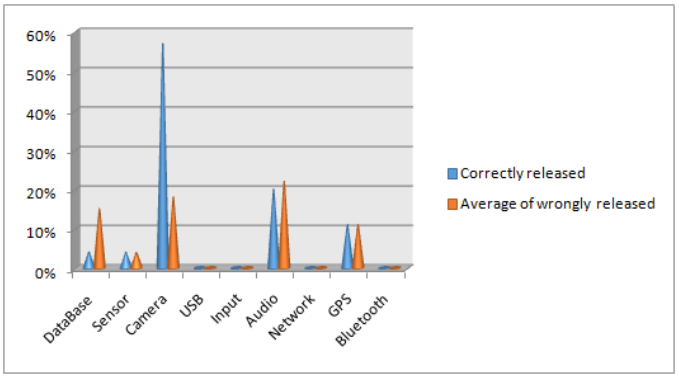


FIGURE 5.8: Column Chart: Distribution of correctly/wrongly released system's resources

Our findings of the correctly/wrongly usage's percentages of the system's resource as shown in Table 5.7 and Figures 5.7 and 5.8:

- *Database* resource should be acquired at onCreate() and released at onPause() methods [17]. Our result shows that about 85% activities used onCreate() to acquire *Database* resource and 4% of activities used onPause() to release *Database* resource correctly. However, the average of wrongly acquired is equal to 25%. It includes 67% of activities used OTHER method, 15% used onDestroy() and 8% used onResume() or onPause() to acquire *Database* resource. Additionally, the average of wrongly released is equal to 15% of activities used onDestroy() to release *Database* resource.

- *Sensor* resource should be acquired at onResume() and released at onPause() [17]. Our result shows that about 4% of activities used onResume() to acquire *Sensor* resource and 4% of activities used onPause() to release *Sensor* resource correctly. However, the average of wrongly acquired is equal to 52%. It includes 100% of activities used onCreate() method and 4% used OTHER to acquire *Sensor* resource. Additionally, the average of wrongly released is equal to 4%. It includes 4% of activities used onCreate() method, and also 4% used OTHER to release *Sensor* resource.

- *Camera* resource should be acquired at onResume() and released at onPause() [17]. Our result shows that about 33% of activities used onResume() to acquire *Camera* resource and 57% of activities used onPause () to release *Camera* resource correctly. However, the average of wrongly acquired is equal to 35%. It includes 81% of activities used OTHER method, 19% used onCreate() and 5% used onStart() to acquire *Camera* resource. Additionally, the average of wrongly released is equal to 13%. It includes 14% of activities used onDestroy() and 57% used OTHER method to release *Camera* resource.
- *USB* should be acquired at onResume() and released at onPause() [17]. Our result showed there are no occurrences of acquired or released *USB* resource in our dataset.
- *Input* resource should be acquired at onCreate() and released at onPause() [17]. Our result showed that about 33% of activities used onResume() to acquire *Input* resource. However, there are no occurrences of released *Input* resource in our dataset.
- *Audio* resource should be acquired at onCreate() and released at onPause() [17]. Our result showed that there are no occurrences of acquired *Audio* resource at onCreate() in our dataset, and 20% of activities used onPause() to release *Audio* resource correctly. However, the average of wrongly released is equal to 30%. It includes 47% of activities used onCreate() method,7% used onDestroy() and 13% used OTHER to release *Audio* resource.
- *Network* resource should be acquired at onCreate() and released at onPause() [17]. Our result showed that there are no occurrences of the acquired or released *Network* resource in our dataset.
- *GPS* resource should be acquired at onCreate() and released at onPause() [17].

Our result showed that about 11% of activities used onCreate() to acquired and also11% of activities used onPause() released *GPS* resource correctly. However, the average of wrongly acquired is equal to 11%. It includes 11% of activities used onResume() method and 11% used OTHER to acquire *GPS* resource. Additionally, the average of wrongly released is equal to 11%. It includes 11% of activities used onCreate()method and 11% used OTHER to release *GPS* resource.

- *Bluetooth* resource should be acquired at onCreate() and released at onPause() [17]. Our result showed that about 40% of activities used onCreate() to acquire *Bluetooth* resource correctly and there are no occurrences of released *Bluetooth* resource at onPause() method. However, the average of wrong acquired is equal to 25%. It includes 40% of activities 40% used OTHER method , 20% used onPause() and onResume() to acquire *Bluetooth* resource.

5.2.2.4 Correctly/Wrongly acquired and released system's resources over the apps categories

In this section, the percentages of correctly/wrongly acquired and released for each system's resources in relation to the app categories was shown. For each system's resource, the percentages were computed according to app categories distribution.

5.2.2.4.1 Camera system's resource *Camera* is a system's resource that is shared by apps on Android mobile devices [40]. It provides APIs that's offered *Camera* features available on mobile devices. This feature allowing user to capture pictures and videos in Android apps. Android apps can make use of *Camera* resource after getting an instance of *Camera* APIs. These APIs are :

- android.hardware.camera2: This API used for taking pictures and video in apps which used *Camera* system's resource. It works on Android 5.0 and greater versions. It uses open() to create an instance of the *Camera* or startPreview() methods to start *Camera* preview. Also, it uses release() to release the *Camera* resource or stopPreview() to stop *Camera* preview.
- android.hardware.camera: This is the old Camera API which used to control *Camera* system's resource.
- android.hardware.camera.CameraDevice or android.hardware.camera2.CameraDevice: These are implementation of a single Camera that connected to an Android device, which allowing for fine-grain control of image capture. It uses openCamera() or onOpened() methods to to open/acquire connection to *Camera* resource. Also, They use close() or onClosed() to release *Camera* resource.
- android.hardware.camera.CameraManager or android.hardware.camera2.CameraManager: These are system service managers which use for detecting, characterizing, and connecting to CameraDevices.They use openCamera() or onOpened() methods to open/acquire connection to *Camera* resource. Also, its use close() or onClosed() to release *Camera* resource.

(1) Acquiring Camera system's resource:

Android apps which use an instance of *Camera* resource to create the camera object or access a particular hardware camera when the app starts using it [40]. Apps must acquire/open the *Camera* resource inside onResume() callback method. The process of opening/acquiring may take a long time to finish on some mobile devices. It is best to call these process camera from a worker thread to prevent blocking the main app UI thread. For example, in case a developer uses an instance *Camera* resource

API, the Camera.startPreview() method will be used as shown in the code bellow:

```
public class CameraActivity extends Activity {
    private Camera mCamera;
  @Override
  protected void onResume() {
        super.onResume();
        openCamera(); // open the camera immediately on resume event
    }
  private void openCamera(){
     if (mCamera == null) {
        mCamera = getCameraInstance();
    }
    mCamera.startPreview();    // start the camera for the app
   }
}
```

After analyzing 5577 Android activities, the acquired *Camera* system's resource percentages result were obtained as shown in the following Table 5.8.

TABLE 5.8: Distribution of acquired Camera system's resource

Category	onCreate()	onStart()	onResume()	onPause()	onStop()	onDestroy()	onRestart()	OTHER
Connectivity	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Development	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Games	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Graphics	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Internet	(2) 10%	(1) 5%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(2) 10%
Money	(1) 5%	(0) 0%	(5) 24%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(6) 29%
Multimedia	(1) 5%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(1) 5%
Navigation	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Phone & SMS	(0) 0%	(0) 0%	(2) 10%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(6) 29%
Reading	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Science & Education	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(1) 5%
Security	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Sports & Health	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
System	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(1) 5%
Theming	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Time	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Writing	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%

Table 5.8 shows the distribution of the acquired *Camera* across the app categories. Each value represents the count and percentage of occurrence of the acquired *Camera* over 21 existence of *Camera* system's resource inside 5577 activities. The result was represented in Bar Chart in Figure 5.9.

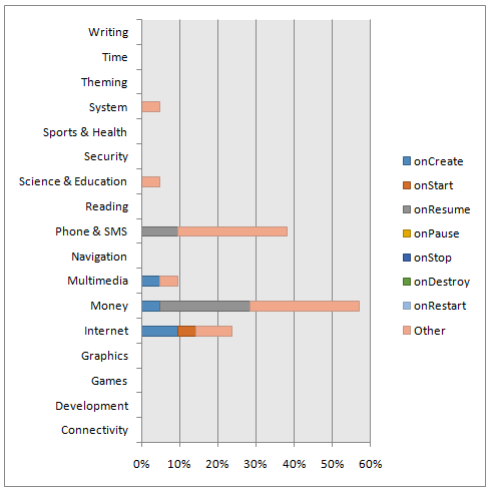


FIGURE 5.9: Bar Chart: Distribution of acquired Camera resource over the app Categories

As shown in Figure 5.9, the *Money* category is the most frequent use for acquiring *Camera* followed by *Phone & SMS* then *Internet*. However, in order to give our indication, the average of wrongly acquired was competed to compare it with the correctly acquired result as shown in Table 5.9.

TABLE 5.9: Distribution of correctly/wrongly acquired Camera system's resource

Category	correctly acquired	Average of wrongly acquired
Connectivity	0% on onResume()	0%
Development	0% on onResume()	0%
Games	0% on onResume()	0%
Graphics	0% on onResume()	0%
Internet	0% on onResume()	8%
Money	24% on onResume()	17%
Multimedia	0% on onResume()	5%
Navigation	0% on onResume()	0%
Phone & SMS	10% on onResume()	29%
Reading	0% on onResume()	0%
Science & Education	0% on onResume()	5%
Security	0% on onResume()	0%
Sports & Health	0% on onResume()	0%
System	0% on onResume()	5%
Theming	0% on onResume()	0%
Time	0% on onResume()	0%
Writing	0% on onResume()	0%
Total Average	33% on onResume()	68%

Table 5.9 was represented in Figure 5.10.

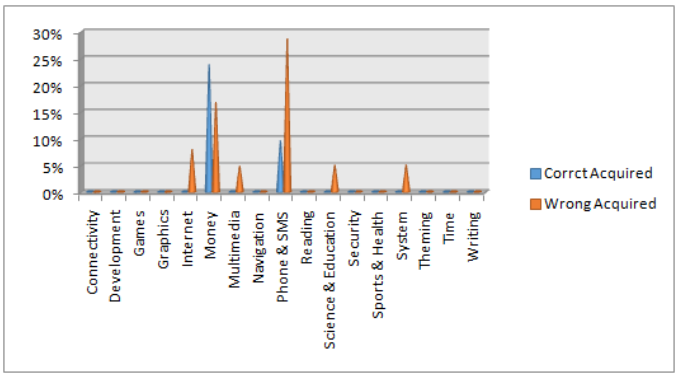


FIGURE 5.10: Column Chart: Distribution of correctly/wrongly acquired Camera resource over the app categories

The compared results in Figure 5.10 showed that:

- The percentage of correctly acquired on onResume() callback method for the *Camera* is 24% at *Money* category, whereas the wrongly percentage is 17%.

- The percentage of correctly acquired on onResume() callback method for the *Camera* is 10% at *Phone & SMS*, whereas the wrongly percentage is 29%.
- The percentages of wrongly acquired of the *Camera* are 8% at *Internet*, and 5%s at *Multimedia, Science & Education and system* categories whereas there are no correctly acquired.
- In the remaining categories, there are no occurrence for acquired *Camera* resource.
- The total average of the correctly acquired percentages in onResume() callback method for the *Camera* in all categories is 33%, whereas the total of the wrongly acquired percentages is 68%.

(2) Releasing camera system's resource:

Android apps use an instance of *Camera* resource must be particularly careful to release the camera object when the app stops using it [40]. This occurs as soon as an app -activity- is paused , activity calls onPause() callback method. So, If Android apps does not properly release the *Camera* resource inside onPause(), all features attempts to access the *Camera* inside the apps and other apps will fail, throw a RuntimeException and may cause these to be shut down. For example, in case a developer uses an instance of the *Camera*, the Camera.release() method will be used as shown in the code bellow:

```
public class CameraActivity extends Activity {

    private Camera mCamera;

    @Override

    protected void onPause() {

        super.onPause();

        releaseCamera();   //release the camera immediately on pause event

    }

    private void releaseCamera(){

        if (mCamera != null){

            mCamera.release();     // release the camera for other apps

            mCamera = null;

        }

}
```

After analyzing 5577 Android activities, the releasing *Camera* system's resource percentages result were obtained as shown in the following Table 5.10.

TABLE 5.10: Distribution of released Camera system's resource

Category	onCreate()	onStart()	onResume()	onPause()	onStop()	onDestroy()	onRestart()	OTHER
Connectivity	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(1) 5%	(0) 0%	(0) 0%
Development	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Games	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Graphics	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Internet	(1) 5%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(1) 5%
Money	(0) 0%	(0) 0%	(0) 0%	(8) 38%	(0) 0%	(1) 5%	(0) 0%	(0) 0%
Multimedia	(0) 0%	(0) 0%	(0) 0%	(2) 10%	(1) 5%	(1) 5%	(0) 0%	(2) 10%
Navigation	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Phone & SMS	(0) 0%	(0) 0%	(2) 10%	(2) 10%	(0) 0%	(0) 0%	(0) 0%	(6) 29%
Reading	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Science & Education	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(1) 5%
Security	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Sports & Health	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
System	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(2) 10%
Theming	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Time	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Writing	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%

Table 5.10 shows the distribution of the released *Camera* across the app categories. Each value represents the count and percentage of occurrence of the released *Camera*

over 21 existence of *Camera* system's resource inside 5577 activities. The result was represented in Bar Chart in Figure 5.11.

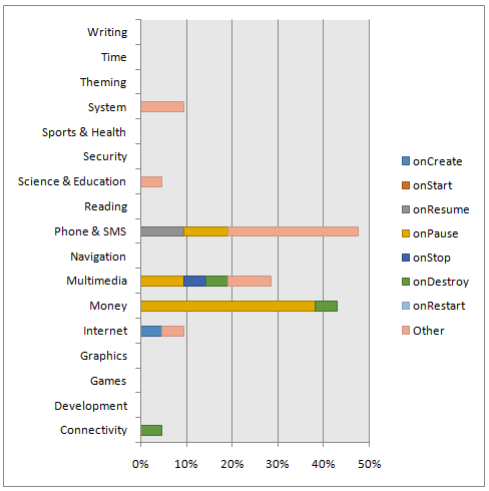


FIGURE 5.11: Bar Chart: Distribution of released Camera resource over the app categories

As shown in Figure 5.11, the *Phone & SMS* category is the most frequent use for releasing *Camera* followed by *Money* then *Multimedia*. However, in order to give our indication, the average of wrongly released was competed to compare it with the correctly released result as shown in Table 5.11.

TABLE 5.11: Distribution of correctly/wrongly released Camera system's resource

Category	correctly released	Average of wrongly released
Connectivity	0% on onPause()	5%
Development	0% on onPause()	0%
Games	0% on onPause()	0%
Graphics	0% on onPause()	0%
Internet	0% on onPause()	5%
Money	38 % on onPause()	5%
Multimedia	10% on onPause()	5%
Navigation	0% on onPause()	0%
Phone & SMS	10% on onPause()	19%
Reading	0% on onPause()	0%
Science & Education	0% on onPause()	5%
Security	0% on onPause()	0%
Sports & Health	0% on onPause()	0%
System	0% on onPause()	10%
Theming	0% on onPause()	0%
Time	0% on onPause()	0%
Writing	0% on onPause()	0%
Total Average	57% on onPause()	53%

Table 5.11 was represented in Figure 5.12.

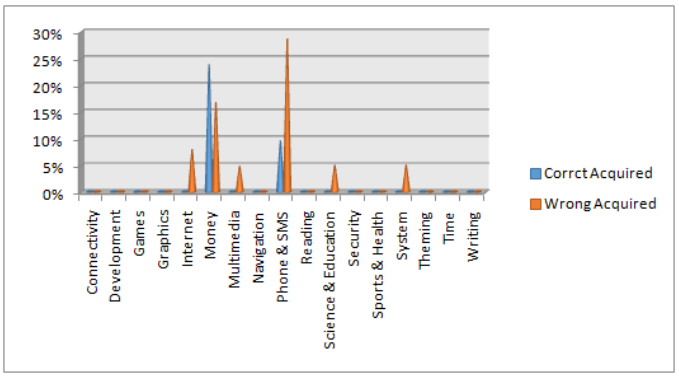


FIGURE 5.12: Column Chart: Distribution of correctly/wrongly released Camera resource over the app categories

The compared results as shown in Figure 5.12 showed that:

- The percentage of correctly released on onPause() callback method for the *Camera* is 10% at *Phone & SMS* category, whereas the wrongly percentage is 19%.

- The percentage of correctly released on onPause() callback method for the *Camera* is 38% at *Money* category, whereas the wrongly percentage is 5%.
- The percentage of correctly released on onPause() callback method for the *Camera* is 10% at *Multimedia* category, whereas the wrongly percentage is 5%.
- The percentages of wrongly released for the *Camera* are 10% at *System* and 5%s at *Science & Education, Internet and connectivity* categories, whereas there are no correctly released.
- In the remaining categories, there are no occurrence for released *Camera* resource.
- The total average of the correctly released percentages on onPause() callback method for the *Camera* in all categories is 57%, whereas the total of the wrongly released percentages is 53%.

5.2.2.4.2 Database System's Resource *Database* system's resource was implemented to explore data returned through a content provider [41]. Android apps can use *Database* resource after getting an instance of SQLite APIs. These APIs are:

- android.database.sqlite.SQLiteDatabase: This API offers developers, accessibility to manage SQLite database and perform common database management tasks such as create, delete, execute SQL commands. It uses openDatabase() or openOrCreateDatabase() methods to open a connection to *Database* resource.
- android.database.sqlite.SQLiteClosable: This API used to close an object created from a SQLiteDatabase. Also, it uses close(), releaseMemory(), releaseReference() and releaseReferenceFromContainer() methods to close *Database* connection through releasing a reference to the object.

(1) Acquiring the database system recourse:

Android apps which use an instance of *Database* resource to create the SQLite object when the app starts using it [41]. As our study of database resource, Apps must acquire/open *Database* resource inside onCreate() callback method [42]. This help ensuring that don't have to create a new on each resume and also have only one *Database* when an Activity is resumed and paused. For example, in case a developer uses an instance of the SQLiteDatabase API, the SQLiteDatabase.openDatabase(...) method will be used as shown in the code bellow:

```
public class DatabaseActivity  extends  Activity  {
    private  SQLiteDatabase  mDatabase;
    @Override
   protected  void  onCreate(){
     super.onCreate();
     mDatabase  =  SQLiteDatabase.create(null);
     mDatabase  =  SQLiteDatabase.openDatabase(...);  // open  the  database immediately  on  create  event
  }
}
```

After analyzing 5577 Android activities, the acquiring *Database* system's resource percentages result were obtained as shown in the following Table 5.12.

TABLE 5.12: Distribution of acquired Database system's resource

Category	onCreate()	onStart()	onResume()	onPause()	onStop()	onDestroy()	onRestart()	OTHER
Connectivity	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Development	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Games	(4) 8%	(0) 0%	(1) 2%	(2) 4%	(0) 0%	(4) 8%	(0) 0%	(3) 6%
Graphics	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Internet	(3) 6%	(0) 0%	(1) 2%	(0) 0%	(0) 0%	(2) 4%	(0) 0%	(3) 6%
Money	(8) 15%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Multimedia	(17) 33%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(13) 25%
Navigation	(1) 2%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(1) 2%
Phone & SMS	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Reading	(1) 2%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Science & Education	(2) 4%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(5) 10%
Security	(4) 8%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(4) 8%
Sports & Health	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
System	(2) 4%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(2) 4%	(0) 0%	(1) 2%
Theming	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Time	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Writing	(2) 4%	(0) 0%	(2) 4%	(2) 4%	(0) 0%	(0) 0%	(0) 0%	(5) 10%

Table 5.12 shows the distribution of the acquired *Database* across the app categories. Each value represents the count and percentage of occurrence of the acquired *Database* over 52 existence of *Database* system's resource inside 5577 activities. The result was represented in Bar Chart in Figure 5.13.

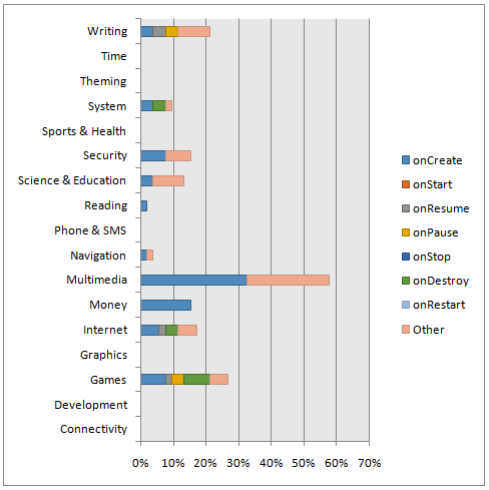


FIGURE 5.13: Bar Chart: Distribution of acquired Database resource over the app categories

As shown in Figure 5.13, the *Multimedia* category is the most frequent use for acquiring *Database* followed by *Games* then *Writing*. However, in order to give our indication, the average of wrongly acquired was competed to compare it with the correctly acquired result as shown in Table 5.13.

TABLE 5.13: Distribution of correctly/wrongly acquired Database system's resource

Category	correctly acquired	Average of wrongly acquired
Connectivity	0% on onCreate()	0%
Development	0% on onCreate()	0%
Games	8% on onCreate()	5%
Graphics	0% on onCreate()	0%
Internet	6% on onCreate()	4%
Money	15% on onCreate()	0%
Multimedia	33% on onCreate()	25%
Navigation	2% on onCreate()	2%
Phone & SMS	0% on onCreate()	0%
Reading	2% on onCreate()	0%
Science & Education	4% on onCreate()	10%
Security	8% on onCreate()	8%
Sports & Health	0% on onCreate()	0%
System	4% on onCreate()	3%
Theming	0% on onCreate()	0%
Time	0% on onCreate()	0%
Writing	4% on onCreate()	6%
Total Average	85% on onCreate()	62%

Table 5.13 was represented in figure 5.14.

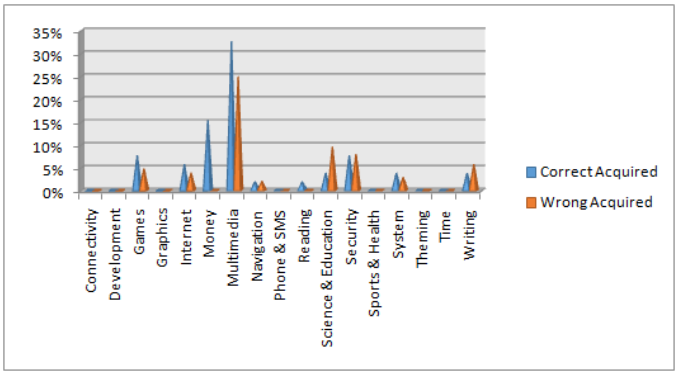


FIGURE 5.14: Column Chart: Distribution of correctly/wrongly acquired Database resource over the app categories

The compared results in Figure 5.14 showed that:

- The percentage of correctly acquired on onCreate() callback method for the *Database* is 33% at *Multimedia* category, whereas the wrongly percentage is 25%.
- The percentage of correctly acquired on onCreate() callback method for the *Database* is 8% at *Games*, whereas the wrongly percentage is 5%.
- The percentage of correctly acquired on onCreate() callback method for the *Database* is 6% at *Internet*, whereas the wrongly percentage is 4%.
- The percentage of correctly acquired on onCreate() callback method for the *Database* is 2% at *Navigation*, whereas the wrongly percentage is also 2%.
- The percentage of correctly acquired on onCreate() callback method for the *Database* is 4% at *Science & Education*, whereas the wrongly percentage is 10%.
- The percentage of correctly acquired on onCreate() callback method for the *Database* is 8% at *Security*, whereas the wrongly percentage is also 8%
- The percentage of correctly acquired on onCreate() callback method for the *Database* is 4% at *System*, whereas the wrongly percentage is 3%.
- The percentage of correctly acquired on onCreate() callback method for the *Database* is 4% at Writing, whereas the wrongly percentage is 10%.
- The percentages of correctly acquired of the *Database* are 15% at *Money* and 2% at *Reading*, whereas there are no wrongly acquired.
- In the remaining categories, there are no occurrence for acquired *Database* resource.
- The total average of the correctly acquired percentages on onCreate() callback method for *Database* in all categories is 85%, whereas the total of the wrongly acquired percentages is 62%.

(2) Releasing the database system recourse:

Android apps use an instance of SQLiteDatabase API must be particularly careful to release the database object when the app stops using it [42]. Developers use onPause() callback method to close *Database* connection. For example, in case a developer uses an instance of the SQLiteDatabase API. the SQLiteDatabase.close() method will be used as shown in the code bellow:

```
public class DatabaseActivity extends Activity {

    private SQLiteDatabase mDatabase;

    @Override

    protected void  onPause()  {

         super.onPause(); ;

        mDatabase.close(); // open the database immediately on pause event

      }

}
```

After analyzing 5577 Android activities, the releasing *Database* system's resource percentages result were obtained as shown in the following Table 5.14.

TABLE 5.14: Distribution of released Database system's resource

Category	onCreate()	onStart()	onResume()	onPause()	onStop()	onDestroy()	onRestart()	OTHER
Connectivity	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Development	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Games	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(4) 8%	(0) 0%	(0) 0%
Graphics	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Internet	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(2) 4%	(0) 0%	(0) 0%
Money	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Multimedia	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Navigation	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Phone & SMS	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Reading	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Science & Education	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Security	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Sports & Health	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
System	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(2) 4%	(0) 0%	(0) 0%
Theming	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Time	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Writing	(0) 0%	(0) 0%	(0) 0%	(2) 4%	(0) 0%	(0) 0%	(0) 0%	(0) 0%

Table 5.14 shows the distribution of the released *Database* across the app categories. Each value represents the count and percentage of occurrence of the released *Database* over 52 existence of *Database* system's resource inside 5577 activities. The result was represented in Bar Chart in Figure 5.15.

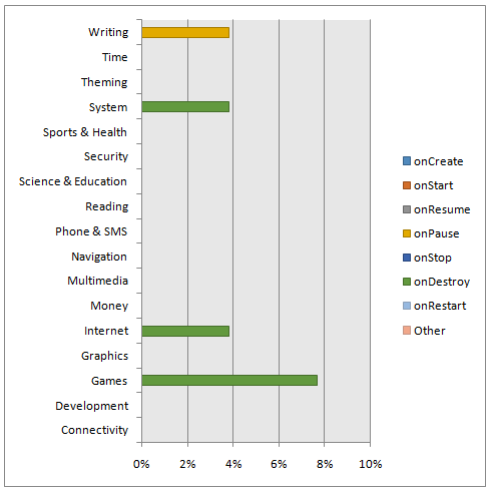


FIGURE 5.15: Bar Chart: Distribution of released Database resource over the app categories

As shown in Figure 5.15, the *Games* category is the most frequent use for releasing *Database* followed by *System* and *Internet*. However, in order to give our indication, the average of wrongly released was competed to compare it with the correctly released result as shown in Table 5.15.

TABLE 5.15: Distribution of correctly/wrongly released Database system's resource

Category	correctly released	Average of wrongly released
Connectivity	0% on onPause()	0%
Development	0% on onPause()	0%
Games	0% on onPause()	8%
Graphics	0% on onPause()	0%
Internet	0% on onPause()	4%
Money	0% on onPause()	0%
Multimedia	0% on onPause()	0%
Navigation	0% on onPause()	0%
Phone & SMS	0% on onPause()	0%
Reading	0% on onPause()	0%
Science & Education	0% on onPause()	0%
Security	0% on onPause()	0%
Sports & Health	0% on onPause()	0%
System	0% on onPause()	4%
Theming	0% on onPause()	0%
Time	0% on onPause()	0%
Writing	4% on onPause()	0%
Total Average	4% on onPause()	16%

Table 5.15 was represented in Figure 5.16.

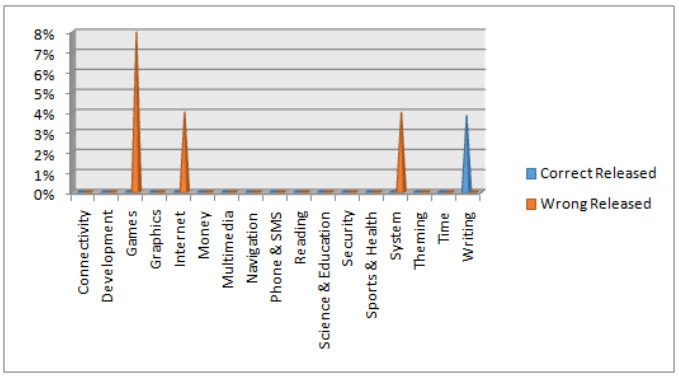


FIGURE 5.16: Column Chart: Distribution of correctly/wrongly released Database resource over the app categories

The compared results in Figure 5.16 showed that:

- The percentages of wrongly released of the *Database* on onPause() callback method are 8% at *Games*, 4% at , *Internet* and 4% at and *System*, whereas there are no correctly released.

- The percentage of wrongly released of the *Database* is 4% at *Writing*, whereas there are no correctly released.
- In the remaining categories, there are no occurrence for released *Database* resource.
- The total average of the correctly released percentages on onPause() callback method for the *Database* in all categories is 4%, whereas the total of the wrongly released percentages is 16%.

5.2.2.4.3 Sensor System's Resource *Sensor* system's resource is built in power mobile apps to compute motion , orientation and other environmental condition [43]. Developers can access to *Sensor* resource thorough the Sensor framework "android.hardware.Sensor".This framework helps developers perform a variety of sensor tasks such as deciding available sensors on a device, decide capabilities of the sensor's, such as power requirements, maximum range, resolution, and manufacturer, acquire sensor raw data, monitor sensor change through register and unregister sensor event listeners.

The Sensor framework uses to create an instance of *Sensor* resource and let developers determine the capabilities of a sensor [43]. It includes also "android.hardware.SensorManager" API which lets developer access the device's sensors by creating instance of sensor service. Its various methods for accessing and listing sensors, registering and unregistering sensor event listeners, and acquiring orientation information. These methods are registerListener(), getsystemservice () or start() to acquire/open *Sensor* resource and unregisterListener() or stop() to release *Sensor* resource.

(1) Acquiring Sensor system's resource:

Android apps which use an instance of *Sensor* resource to create the sensor object or access a particular hardware sensor when the app starts using it [43]. Apps must

acquire/open the *Sensor* resource inside onResume() callback method. For example, in case a developer uses an instance of the SensorManager API, the SensorManager.registerListener(..) method will be used as shown in the code bellow:

```
public class SensorActivity extends Activity implements SensorEventListener {
    private final SensorManager mSensorManager;
    private final Sensor mAccelerometer;

    public SensorActivity() {
        mSensorManager = (SensorManager)getSystemService(SENSOR_SERVICE);
        mAccelerometer = mSensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER);
    }

    protected void onResume() {
        super.onResume();
        mSensorManager.registerListener(this, mAccelerometer, SensorManager.SENSOR_DELAY_NORMAL);
    }
  }
```

After analyzing 5577 Android activities, the acquiring *Sensor* system's resource percentages result were obtained as shown in the following Table 5.16.

TABLE 5.16: Distribution of acquired Sensor system's resource

Category	onCreate()	onStart()	onResume()	onPause()	onStop()	onDestroy()	onRestart()	OTHER
Connectivity	(2) 8%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Development	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Games	(5) 19%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Graphics	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Internet	(1) 4%	(0) 0%	(1) 4%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Money	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Multimedia	(3) 12%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Navigation	(10) 38%	(0) 0%	(1) 4%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(1) 4%
Phone & SMS	(2) 8%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Reading	(2) 8%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Science & Education	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Security	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Sports & Health	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
System	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Theming	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Time	(1) 4%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Writing	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%

Table 5.16 shows the distribution of the acquired *Sensor* across the app categories. Each value represents the count and percentage of occurrence of the acquired *Sensor*

over 26 existence of *Sensor* system's resource inside 5577 activities. The result was represented in Bar Chart in Figure 5.17.

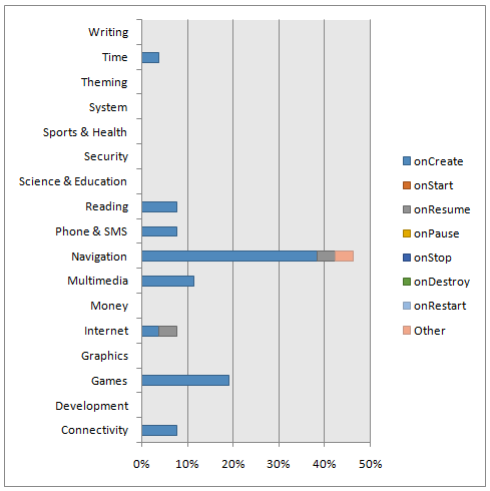


FIGURE 5.17: Bar Chart: Distribution of acquired Sensor resource over the app categories

As shown in Figure 5.17, the *Navigation* category is the most frequent use for acquiring *Sensor* followed by *Games*. However, in order to give our indication, the average of wrongly acquired was competed to compare it with the correctly acquired result as shown in Table 5.17.

TABLE 5.17: Distribution of correctly/wrongly acquired Sensor system's resource

Category	correctly acquired	Average of wrongly acquired
Connectivity	0% on onResume()	8%
Development	0% on onResume()	0%
Games	0% on onResume()	19%
Graphics	0% on onResume()	0%
Internet	4% on onResume()	4%
Money	0% on onResume()	0%
Multimedia	0% on onResume()	12%
Navigation	4% on onResume()	21%
Phone & SMS	0% on onResume()	8%
Reading	0% on onResume()	8%
Science & Education	0% on onResume()	0%
Security	0% on onResume()	0%
Sports & Health	0% on onResume()	0%
System	0% on onResume()	0%
Theming	0% on onResume()	0%
Time	0% on onResume()	4%
Writing	0% on onResume()	0%
Total average	8% on onResume()	83%

Table 5.17 was represented in Figure 5.18.

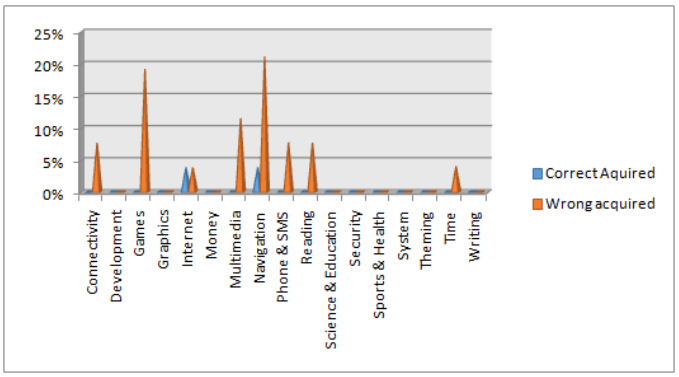


FIGURE 5.18: Column Chart: Distribution of correctly/wrongly acquired Sensor resource over the app categories

The compared results in Figure 5.18 showed that:

- The percentage of correctly acquired on onResume() callback method for the *Sensor* is 4% at *Navigation* category, whereas the wrongly percentage is 21%.

- The percentage of correctly acquired on onResume() callback method for the *Sensor* is 4% at *Internet*, whereas the wrongly percentage is also 4%.
- The percentages of wrongly acquired of the *Sensor* are 8% at *Connectivity*, 19% at *Games*, 12% at *Multimedia*, 8% at *Phone & SMS*, 8% at *Reading* and 4% at *Time* categories, whereas there are no correctly acquired.
- In the remaining categories, there are no occurrence for acquired *Sensor* resource.
- The total average of the correctly acquired percentages on onResume() callback method for the *Sensor* in all categories is 8%, whereas the total of the wrongly acquired percentages is 83%.

(2) Releasing Sensor system's resource:

Android apps used an instance of *Sensor* resource must be particularly careful to release the sensor object when the app stops using it [43]. This occurs as soon as an app -activity- is paused , activity calls onPause() callback method. So, If Android apps does not properly release the *Sensor* resource inside onPause() may cause these to be shut down. For example, in case a developer uses an instance of the SensorManager API, the SensorManager.unregisterListener(this) method will be used as shown in the code bellow:

```
public class SensorActivity extends Activity implements SensorEventListener {
    private final SensorManager mSensorManager;
    private final Sensor mAccelerometer;

    public SensorActivity() {
        mSensorManager = (SensorManager)getSystemService(SENSOR_SERVICE);
        mAccelerometer = mSensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER);
    }

    protected void onPause() {
        super.onPause();
        mSensorManager.unregisterListener(this);
    }
}
```

After analyzing 5577 Android activities, the releasing *Sensor* system's resource percentages result were obtained as shown in the following Table 5.18.

TABLE 5.18: Distribution of released Sensor system's resource

Category	onCreate()	onStart()	onResume()	onPause()	onStop()	onDestroy()	onRestart()	OTHER
Connectivity	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Development	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Games	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Graphics	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Internet	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Money	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Multimedia	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Navigation	(1) 4%	(0) 0%	(0) 0%	(1) 4%	(0) 0%	(0) 0%	(0) 0%	(1) 4%
Phone & SMS	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Reading	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Science & Education	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Security	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Sports & Health	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
System	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Theming	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Time	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Writing	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%

Table 5.18 shows the distribution of the released *Sensor* across the app categories. Each value represents the count and percentage of occurrence of the released *Sensor* over 26 existence of *Sensor* system's resource inside 5577 activities. The result was represented in Bar Chart in Figure 5.19.

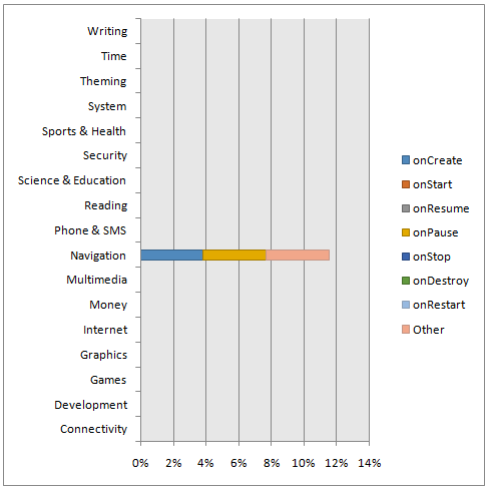


FIGURE 5.19: Bar Chart: Distribution of released Sensor resource over the app categories

As shown in Figure 5.19, the *Navigation* category is the most frequent use for releasing *Sensor*. However, In order to give our indication, the average of wrongly acquired was competed to compare it with the correctly released result as shown in Table 5.19.

TABLE 5.19: Distribution of correctly/wrongly released Sensor system's resource

Category	correctly released	Average of wrongly released
Connectivity	0% on onPause()	0%
Development	0% on onPause()	0%
Games	0% on onPause()	0%
Graphics	0% on onPause()	0%
Internet	0% on onPause()	0%
Money	0% on onPause()	0%
Multimedia	0% on onPause()	0%
Navigation	4% on onPause()	4%
Phone & SMS	0% on onPause()	0%
Reading	0% on onPause()	0%
Science & Education	0% on onPause()	0%
Security	0% on onPause()	0%
Sports & Health	0% on onPause()	0%
System	0% on onPause()	0%
Theming	0% on onPause()	0%
Time	0% on onPause()	0%
Writing	0% on onPause()	0%
Total Average	4% on onPause()	4%

Table 5.19 was represented in Figure 5.20.

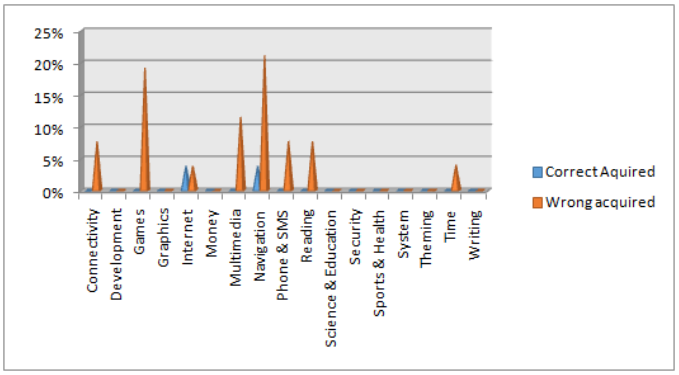


FIGURE 5.20: Column Chart: Distribution of correctly/wrongly released Sensor resource over the app categories

The compared results as shown in Figure 5.20 showed that:

- The percentage of correctly released on onPause() callback method for the *Sensor* is 4% at *Navigation* category, whereas the wrongly percentage is also 4%.
- In the remaining categories, there are no occurrence for released *Sensor* resource.
- The total average of the correctly released percentages on onPause() callback method for the *Sensor* in all categories is 4%, whereas the total of the wrongly released percentages is also 4% .

5.2.2.4.4 GPS System's Resource *GPS* system's resource is used to define Android location and related services [44]. It allows apps to be smarter and offer good information to the user. Using *GPS* resource developers can provide and acquire the user location. *GPS* resource is most accurate, However its consume the power battery, only works outdoors and take a lot time to produce the location. Android location framework offers "android.location.LocationManager" ApI to give developers access to *GPS* resource [45]. Developers can acquire a connection to GPS services using getSystemService(), requestLocationUpdates(), getGpsStatus() and start() methods. Also, they can release/close *GPS* resource using removeUpdates(), stop(),release()

and cancel().

(1) Acquiring the GPS system's resource:

Android apps which use an instance of LocationManager API to create the *GPS* resource object. In Android apps, developers must acquire/open the *GPS* resource inside onCreate() callback method. For example, in case a developer uses an instance of the LocationManager API, the LocationManager.getSystemService() method will be used as shown in the code bellow:

```
public class GPSActivity extends Activity {
    private LocaionManager locationManager;
    @Override
    protected void onCreate(){
     super.onCreate();
     // Acquire a reference to the system Location Manager
     locationManager=(LocationManager)this.getSystemService(Context.Location.Service);
         // Register the listener with the Location Manager to receive location
         updates
     locationManager.requestLocationUpdates(LocationManager.NETWORK_PROVIDER, 0, 0, locationListener);
}
}
```

After analyzing 5577 Android activities, the acquiring *GPS* system's resource percentages result were obtained as shown in the following Table 5.20.

TABLE 5.20: Distribution of acquired GPS system's resource

Category	onCreate()	onStart()	onResume()	onPause()	onStop()	onDestroy()	onRestart()	OTHER
Connectivity	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Development	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Games	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Graphics	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Internet	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Money	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Multimedia	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Navigation	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Phone & SMS	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Reading	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Science & Education	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Security	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Sports & Health	(1) 11%	(0) 0%	(1) 11%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(1) 11%
System	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Theming	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Time	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Writing	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%

Table 5.20 shows the distribution of the acquired *GPS* across the app categories. Each value represents the count and percentage of occurrence of the acquired *GPS* over 9 existence of *GPS* system's resource inside 5577 activities. The result was represented in Bar Chart in Figure 5.21.

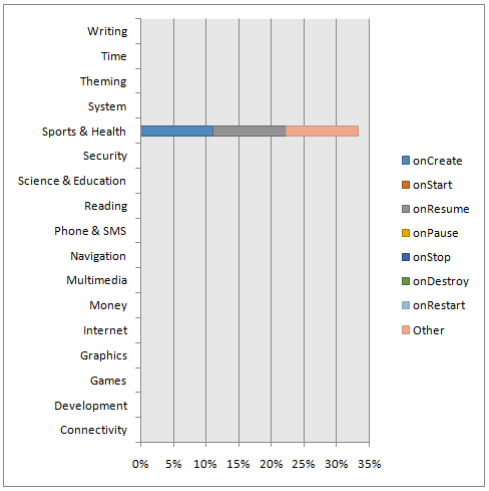


FIGURE 5.21: Bar Chart: Distribution of acquired GPS resource over the app categories

As shown in Figure 5.21, the *Sports & Health* category is the most frequent use for acquiring *GPS*. However, in order to give our indication, the average of wrongly acquired was competed to compare it with the correctly acquired result as shown in Table 5.21.

TABLE 5.21: Distribution of correctly/wrongly acquired GPS system's resource

Category	correctly acquired	Average of wrongly acquired
Connectivity	0% on onCreate()	0%
Development	0% on onCreate()	0%
Games	0% on onCreate()	0%
Graphics	0% on onCreate()	0%
Internet	0% on onCreate()	0%
Money	0% on onCreate()	0%
Multimedia	0% on onCreate()	0%
Navigation	0% on onCreate()	0%
Phone & SMS	0% on onCreate()	0%
Reading	0% on onCreate()	0%
Science & Education	0% on onCreate()	0%
Security	0% on onCreate()	0%
Sports & Health	11% on onCreate()	11%
System	0% on onCreate()	0%
Theming	0% on onCreate()	0%
Time	0% on onCreate()	0%
Writing	0% on onCreate()	0%
Total Average	11% on onCreate()	11%

Table 5.21 was represented in Figure 5.22.

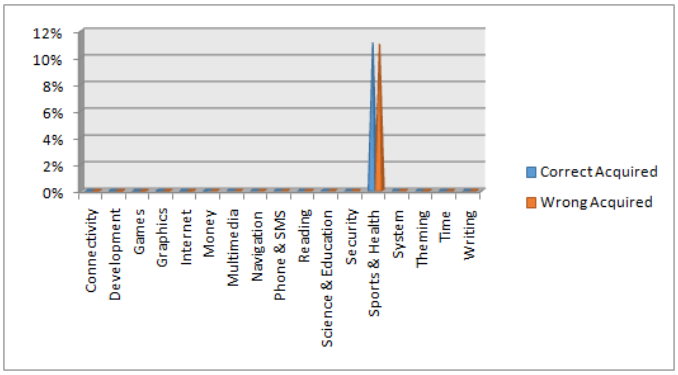


FIGURE 5.22: Column Chart: Distribution of correctly/wrongly acquired GPS resource over the app categories

The compared results in Figure 5.22 showed that:

- The percentage of correctly acquired on onCreate() callback method for the *GPS* is 11% at *Sports & Health* category, whereas the wrongly percentage is also 11%.
- In the remaining categories, there are no occurrence for acquired *GPS* resource.
- The total average of the correctly acquired percentages on onCreate() callback method for the *GPS* in all categories is 11%, whereas the total of the wrongly acquired percentages is also 11%.

(2) Releasing GPS system's resource:

Android apps used an instance of LocationManager API must be particularly careful to release the GPS object when the app stops using it [40]. Developers use onPause() callback method to close *GPS* connection. For example, in case a developer uses an instance of LocationManagera API, the LocationManager.removeUpdates() method will be used as shown in the code bellow:

```
public class GPSActivity extends Activity {

    private LocaionManager locationManager;

    @Override

       public override void OnPause()
       {
           super.OnPause();
           locationManager.RemoveUpdates(this);
       }
   }
}
```

After analyzing 5577 Android activities, the releasing *GPS* system's resource percentages result were obtained as shown in the following Table 5.22.

TABLE 5.22: Distribution of released GPS system's resource

Category	onCreate()	onStart()	onResume()	onPause()	onStop()	onDestroy()	onRestart()	OTHER
Connectivity	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Development	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Games	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Graphics	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Internet	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Money	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Multimedia	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Navigation	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Phone & SMS	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Reading	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Science & Education	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Security	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Sports & Health	(1) 11%	(0) 0%	(0) 0%	(1) 11%	(0) 0%	(1) 11%	(0) 0%	(1) 11%
System	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Theming	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Time	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Writing	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%

Table 5.22 shows the distribution of the released *GPS* across the app categories. Each value represents the count and percentage of occurrence of the released *GPS* over 9 existence of *GPS* system's resource inside 5577 activities. The result was represented in Bar Chart in Figure 5.23.

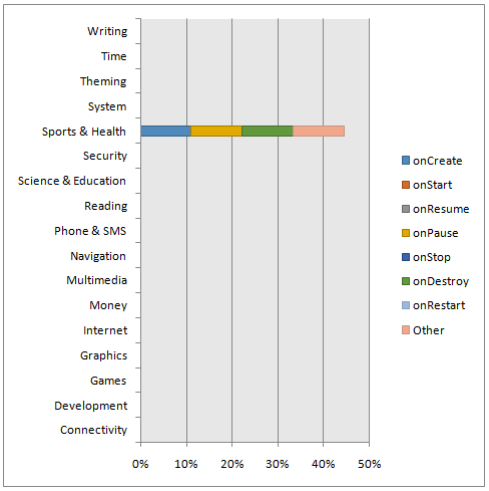


FIGURE 5.23: Bar Char: Distribution of released GPS resource over the app categories

As shown in Figure 5.23, the *Sports & Health* category is the most frequent use for releasing *GPS*. However, in order to give our indication, the average of wrongly released is competed to compare it with the correctly released result as shown in Table 5.23.

TABLE 5.23: Distribution of correctly/wrongly released GPS system's resource

Category	correctly released	Average of wrongly released
Connectivity	0% on onPause()	0%
Development	0% on onPause()	0%
Games	0% on onPause()	0%
Graphics	0% on onPause()	0%
Internet	0% on onPause()	0%
Money	0% on onPause()	0%
Multimedia	0% on onPause()	0%
Navigation	0% on onPause()	0%
Phone & SMS	0% on onPause()	0%
Reading	0% on onPause()	0%
Science & Education	0% on onPause()	0%
Security	0% on onPause()	0%
Sports & Health	11% on onPause()	11%
System	0% on onPause()	0%
Theming	0% on onPause()	0%
Time	0% on onPause()	0%
Writing	0% on onPause()	0%
Total Average	11% on onPause()	11%

Table 5.23 was represented in Figure 5.24.

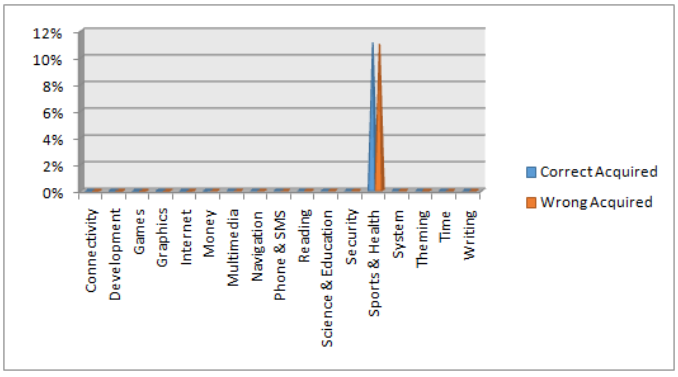


FIGURE 5.24: Column Chart: Distribution of correctly/wrongly released GPS resource over the app categories

The compared results as shown in Figure 5.24 showed that:

- The percentage of correctly released on onPause() callback method for the *GPS* is 11% at *Sports & Health* category, whereas the wrongly percentage is also 11%.
- In the remaining categories, there are no occurrence for acquired *GPS* resource.
- The total average of the correctly released percentages on onPause() callback method for the *GPS* in all categories is 11%, whereas the total of the wrongly released percentages is also 11%.

5.2.2.4.5 Input System's Resource *Input* system's resource is used to provide information about *Input* devices and available key layouts [46]. Input framework provides

android.hardware.input.InputManager API which lets developers access or acquire input resources. Developers can acquire *Input* resource using getsystemservice(), registerInputDeviceListener(), or start() methods. Also, they can release *Input* resource using unregisterInputDeviceListener() or stop() methods.

(1) Acquiring Input system's resource:

Android apps which use an instance of InputManager API to create the input object. Apps must acquire/open the *Input* resource inside onCreate() callback method. For example, in case a developer uses an instance of the InputManager API, the InputManager.getsystemservice() method will be called as shown in the code bellow:

```
public class InputActivity extends Activity {
    private InputManager mInputManager;
    @Override
    protected void onCreate(){
     super.onCreate();
     // Acquire a reference to the system Input Manager
    mInputManager=(InputManager)getSystemService(Context.INPUT_SERVICE);
    mInputManager.registerInputDeviceListener( this, null );
  }
}
```

After analyzing 5577 Android activities, the acquiring *Input* system's resource percentages result were obtained as shown in the following Table 5.24.

TABLE 5.24: Distribution of acquired Input system's resource

Category	onCreate()	onStart()	onResume()	onPause()	onStop()	onDestroy()	onRestart()	OTHER
Connectivity	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Development	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Games	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Graphics	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Internet	(2) 11%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Money	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Multimedia	(1) 6%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Navigation	(1) 6%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Phone & SMS	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Reading	(1) 6%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Science & Education	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Security	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Sports & Health	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
System	(1) 6%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Theming	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Time	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Writing	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%

Table 5.24 shows the distribution of the acquired *Input* across the app categories. Each value represents the count and percentage of occurrence of the acquired *Input* over 18 existence of *Input* system's resource inside 5577 activities. The result was represented in Bar Chart in Figure 5.25.

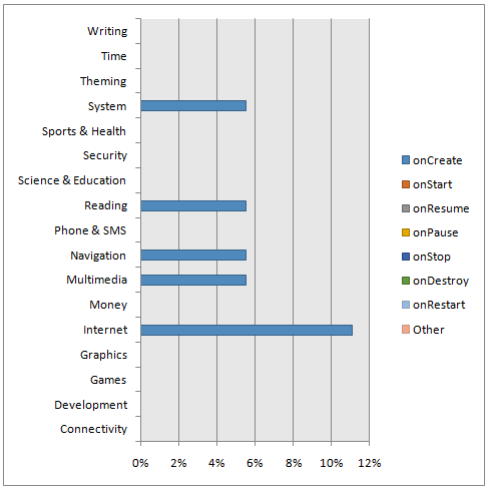


FIGURE 5.25: Bar Chart: Distribution of acquired Input resource over the app categories

As shown in Figure 5.25, *Internet* category is the most frequent use for acquiring *Input*. However, in order to give our indication, the average of wrongly acquired was competed to compare it with the correctly acquired result as shown in Table 5.25.

TABLE 5.25: Distribution of correctly/wrongly acquired Input system's resource

Category	correctlyly acquired	Average of wronglyly acquired
Connectivity	0% on onCreate()	0%
Development	0% on onCreate()	0%
Games	0% on onCreate()	0%
Graphics	0% on onCreate()	0%
Internet	11% on onCreate()	0%
Money	0% on onCreate()	0%
Multimedia	6% on onCreate()	0%
Navigation	6% on onCreate()	0%
Phone & SMS	0% on onCreate()	0%
Reading	6% on onCreate()	0%
Science & Education	0% on onCreate()	0%
Security	0% on onCreate()	0%
Sports & Health	0% on onCreate()	0%
System	6% on onCreate()	0%
Theming	0% on onCreate()	0%
Time	0% on onCreate()	0%
Writing	0% on onCreate()	0%
Total Average	33% on onCreate()	0%

Table 5.25 was represented in Figure 5.26.

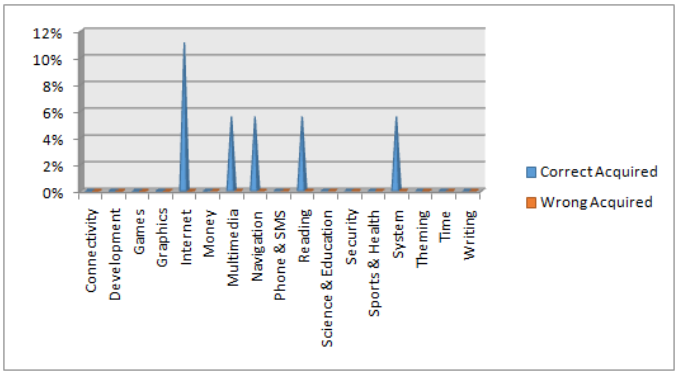


FIGURE 5.26: Column Chart: Distribution of correctly/wrongly acquired Input resource over the app categories

The compared results in Figure 5.26 showed that:

- The percentages of correctly acquired of the *Input* are 11% at *Internet*, and 6%s at *Multimedia*, *Navigation*, *Reading* and *System* categories, whereas there are no wrongly acquired.
- In the remaining categories, there are no occurrence for acquired *Input* resource.
- The total average of the correctly acquired percentages on onCreate() callback method for the *Input* in all categories is 33%, whereas the total of the wrongly acquired percentages is 0%.

(2) Releasing Input system's resource:

Android apps which used an instance of InputManager API must be particularly careful to release the input object when the app stops using it [46]. Developers use onPause() callback method to close input connection. For example, in case a developer uses an instance of the InputManager API, the InputManager.unregisterInputDeviceListener() method will be used as shown in the code bellow:

```
public class InputActivity extends Activity {
    private InputManager mInputManager;
    @Override
    protected void onPause(){
     super.onPause();
     mInputManager.unregisterInputDeviceListener(this);
    }
}
```

After analyzing 5577 Android activities, the releasing *Input* system's resource percentages result were obtained as shown in the following Table 5.26. Table 5.26 shows the distribution of the released *Input* across the app categories. Each value represents the count and percentage of occurrence of the released *Input* over 18 existence of *Input* system's resource inside 5577 activities. It shows that there are no occurrence for released *Input* resource in all app categories.

TABLE 5.26: Distribution of released Input system's resource

Category	onCreate()	onStart()	onResume()	onPause()	onStop()	onDestroy()	onRestart()	OTHER
Connectivity	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Development	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Games	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Graphics	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Internet	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Money	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Multimedia	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Navigation	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Phone & SMS	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Reading	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Science & Education	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Security	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Sports & Health	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
System	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Theming	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Time	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Writing	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%

5.2.2.4.6 Bluetooth System's Resource *Bluetooth* resource provides developers a framework to manage Bluetooth functionality [47]. These functionality such as, scanning for and connecting with devices also managing transformation data between multiple devices. Moreover, *Bluetooth* APIs support low energy and classic kinds. To perform Bluetooth communication using these APIs. *Bluetooth* offers BluetoothManager "android.bluetooth.BluetoothManager" API which can be instance of :

- BluetoothAdapter: This API lets developer perform fundamental Bluetooth tasks, such as initiate device discovery, query a list of bonded devices, instantiate a BluetoothDevice , and create a BluetoothServerSocket to listen for connection requests from other devices, and start a scan for Bluetooth low energy devices. It can used getDefaultAdapter(), enable(), startLeScan(), startDiscovery(), getDefaultAdapter(), getbondeddevices() to acquire connection with Bluetooth resource. Also, stopLeScan(), cancelDiscovery(), closeProfileProxy() to released connection.
- BluetoothDevice , BluetoothServerSocket or BluetoothSocket API use accept() or close() to acquire/release connection to *Bluetooth* resource.

(1) Acquiring Bluetooth system recourse:

Using *Bluetooth* resource, a High level manager BluetoothManager API used to obtain an instance of an BluetoothAdapter and to conduct overall Bluetooth Management [47]. Apps must acquire/open *Bluetooth* resource inside onCreate() callback method. For example, in case a developer uses an instance of the BluetoothAdapter API, the BluetoothAdapter.getDefaultAdapter() method will be used as shown in the code bellow:

```
public class BluetoothActivity extends Activity {
   private BluetoothAdapter mBluetoothAdapter;
   @Override
   protected void onCreate(Bundle savedInstanceState) {
      mBluetoothAdapter = BluetoothAdapter.getDefaultAdapter();
   }
}
```

.

After analyzing 5577 Android activities, the percentages of acquiring *Bluetooth* system's resource result were obtained as shown in the following Table 5.27.

TABLE 5.27: Distribution of acquired Bluetooth system's resource

Category	onCreate()	onStart()	onResume()	onPause()	onStop()	onDestroy()	onRestart()	OTHER
Connectivity	(1) 20%	(1) 20%	(0) 0%	(0) 0%	(0) 0%	(1) 20%	(0) 0%	(2) 40%
Development	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Games	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Graphics	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Internet	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Money	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Multimedia	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Navigation	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Phone & SMS	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Reading	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Science & Education	(1) 20%	(0) 0%	(1) 20%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Security	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Sports & Health	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
System	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Theming	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Time	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Writing	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%

Table 5.27 shows the distribution of the acquired *Bluetooth* across the app categories. Each value represents the count and percentage of occurrence of the acquired *Bluetooth* over 5 existence of *Bluetooth* system's resource inside 5577 activities. The result was represented in bar chart in Figure 5.27.

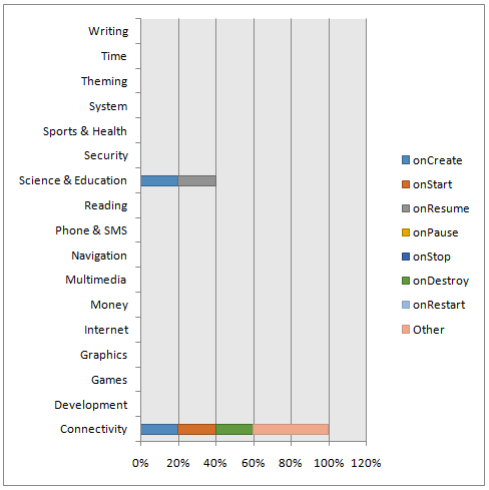


FIGURE 5.27: Bar Chart: Distribution of acquired Bluetooth resource over the app categories

As shown in Figure 5.27, the *Connectivity* and *Science & Education* category are the most frequent use for acquiring *Bluetooth*. However, in order to give our indication, the average of wrongly acquired was competed to compare it with the correctly acquired result as shown in Table 5.28.

TABLE 5.28: Distribution of correctly/wrongly acquired Bluetooth system's resource

Category	correctly acquired	Average of wrongly acquired
Connectivity	20% on onCreate()	27%
Development	0% on onCreate()	0%
Games	0% on onCreate()	0%
Graphics	0% on onCreate()	0%
Internet	0% on onCreate()	0%
Money	0% on onCreate()	0%
Multimedia	0% on onCreate()	0%
Navigation	0% on onCreate()	0%
Phone & SMS	0% on onCreate()	0%
Reading	0% on onCreate()	0%
Science & Education	20% on onCreate()	20%
Security	0% on onCreate()	0%
Sports & Health	0% on onCreate()	0%
System	0% on onCreate()	0%
Theming	0% on onCreate()	0%
Time	0% on onCreate()	0%
Writing	0% on onCreate()	0%
Total Average	40% on onCreate()	47%

Table 5.28 was represented in Figure 5.28.

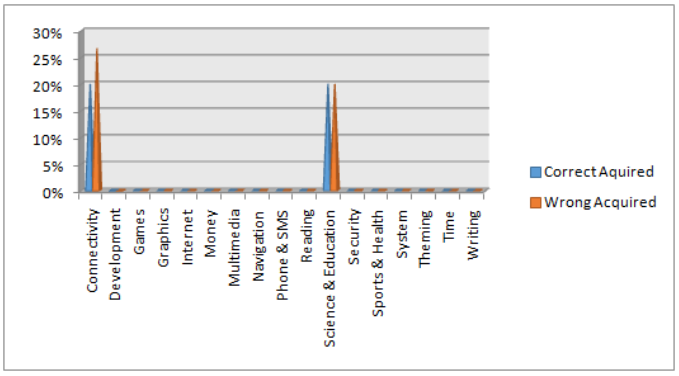


FIGURE 5.28: Column Chart: Distribution of correctly/wrongly acquired Bluetooth resource over the app categories

The compared results in Figure 5.28 showed that:

- The percentage of correctly acquired on onCreate() callback method for the *Bluetooth* is 20% at *Connectivity* category, whereas the wrongly percentage is 27%.
- The percentage of correctly acquired on onCreate() callback method for the *Bluetooth* is 20% at *Science & Education*, whereas the wrongly percentage is also 20%.
- In the remaining categories, there are no occurrence for acquired *Bluetooth* resource.
- The total average of the correctly acquired percentages on onCreate() callback method for the *Bluetooth* in all categories is 40%, whereas the total of the wrongly acquired percentages is 47%.

(2) Releasing Bluetooth system's resource:

Android apps which used an instance of BluetoothAdapter must be particularly careful to release the Bluetooth object when the app stops using [47]. Developers use onPause() callback method to close *Bluetooth* connection. For example, in case a developer uses an instance of the BluetoothAdapter API, the BluetoothAdapter.disable() method will be used as shown in the code bellow:

```
public class BluetoothActivity extends Activity {
   private BluetoothAdapter mBluetoothAdapter;
   @Override
   protected void onPause() {
       super.onPause()
      mBluetoothAdapter.disable()
   }
}
```

After analyzing 5577 Android activities, the percentages of releasing *Bluetooth* system's resource result were obtained as shown in the following Table 5.29. Table 5.10 shows the distribution of the released *Bluetooth* across the app categories. Each value represents the count and percentage of occurrence of the released *Bluetooth* over 5 existence of *Bluetooth* system's resource inside 5577 activities. It shows that there are no occurrence for released *Bluetooth* resource in all app categories.

TABLE 5.29: Distribution of released Bluetooth system's resource

Category	onCreate()	onStart()	onResume()	onPause()	onStop()	onDestroy()	onRestart()	OTHER
Connectivity	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Development	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Games	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Graphics	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Internet	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Money	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Multimedia	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Navigation	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Phone & SMS	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Reading	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Science & Education	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Security	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Sports & Health	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
System	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Theming	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Time	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Writing	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%

5.2.2.4.7 Audio System's Resource *Audio* resource is an API from Media framework

"Android.media" [48]. It provides classes that manage various media interfaces in *Audio* resource. The Media API was used to play or record Audio files. Using AudioManager API "android.media.AudioManager" inside media framework let developers access to volume and ringer mode control. Developers use getSystemService(), registerAudioDeviceCallback() or start() to acquire connection with *Audio* resource. Also, they use stop(), release(), cancel() or unregisterAudioDeviceCallback() to release connection.

(1) Acquiring Audio system recourse:

Android apps which use an instance of AudioManager to create the *Audio* resource object when the app starts using it [48]. Apps must acquire/open the *Audio* resource inside onCreate() callback method. For example, in case a developer uses an instance of the AudioManager API, the AudioManager.getSystemService() method will be used as shown in the code bellow:

```
public class AudioActivity extends Activity  {
   private AudioManager mAudioManager;
   @Override
   protected void onCreate() {
       super. onCreate()
      myAudioManager  = (AudioManager)getSystemService(Context.AUDIO_SERVICE);
   }
}
```

After analyzing 5577 Android activities, the acquired percentages for *Audio* system's resource were obtained as shown in the following Table 5.30.

TABLE 5.30: Distribution of acquired Audio system's resource

Category	onCreate()	onStart()	onResume()	onPause()	onStop()	onDestroy()	onRestart()	OTHER
Connectivity	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Development	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Games	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Graphics	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Internet	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Money	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Multimedia	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Navigation	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Phone & SMS	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Reading	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Science & Education	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Security	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Sports & Health	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
System	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Theming	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Time	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Writing	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%

Table 5.30 shows the distribution of acquired *Audio* percentages across the app categories. Each value represents the count and percentage of occurrence of the acquired *Audio* over 15 existence of *Audio* system's resource inside 5577 activities. The result shows that there are no occurrence for acquired *Audio* resource in the all app categories.

(2) Releasing Audio System's Resource:

Android apps which used an instance of AudioManager resource must be particularly careful to release the *Audio* object when the app stops using it [48]. This occurs as soon as an app -activity- is paused , activity calls onPause() callback method. Developers use onPause() callback method to close *Audio* resource. For example, in case a developer uses an instance of the AudioManager API, the AudioManager.release() method will be used as shown in the code bellow:

```
public class AudioActivity extends Activity  {
   private AudioManager mAudioManager;
   @Override
   protected void onPause() {
       super.onPause()
      myAudioManager.release();
   }
}
```

After analyzing 5577 Android activities, the released percentages for *Audio* system's resource result were obtained as shown in the following Table 5.31.

TABLE 5.31: Distribution of released Audio system's resource

Category	onCreate()	onStart()	onResume()	onPause()	onStop()	onDestroy()	onRestart()	OTHER
Connectivity	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Development	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Games	(3) 20%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(2) 13%
Graphics	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Internet	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Money	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Multimedia	(1) 7%	(0) 0%	(0) 0%	(3) 20%	(0) 0%	(1) 7%	(0) 0%	(0) 0%
Navigation	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Phone & SMS	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Reading	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Science & Education	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Security	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Sports & Health	(2) 13%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
System	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Theming	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Time	(1) 7%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Writing	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%

Table 5.31 shows the distribution of the released *Audio* percentages across the app categories. Each value represents the count and percentage of occurrence of the released *Audio* over 15 existence of Audio system's resource inside 5577 activities. The result was represented in Bar Chart at Figure 5.29.

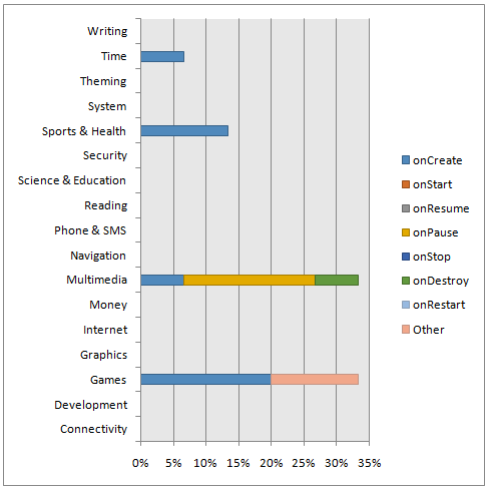


FIGURE 5.29: Bar chart: Distribution of released Audio resource over the app categories

As shown in Figure 5.29, the *Multimedia* category is the most frequent use for releasing Audio methods followed by *Games*. However, in order to give our indication, the average of wrongly released was competed to compare it with correctly released result as shown in Table 5.32.

TABLE 5.32: Distribution of correctly/wrongly released Audio system's resource

Category	Correctly released	Average of wrongly released
Connectivity	0% on onPause()	0%
Development	0% on onPause()	0%
Games	0% on onPause()	17%
Graphics	0% on onPause()	0%
Internet	0% on onPause()	0%
Money	0% on onPause()	0%
Multimedia	20% on onPause()	7%
Navigation	0% on onPause()	0%
Phone & SMS	0% on onPause()	0%
Reading	0% on onPause()	0%
Science & Education	0% on onPause()	0%
Security	0% on onPause()	0%
Sports & Health	0% on onPause()	13%
System	0% on onPause()	0%
Theming	0% on onPause()	0%
Time	0% on onPause()	7%
Writing	0% on onPause()	0%
Total Average	20% on onPause()	43%

Table 5.32 was represented in Figure 5.30.

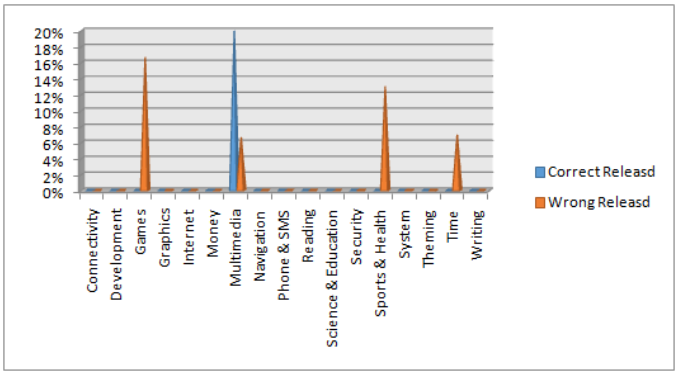


FIGURE 5.30: Column Chart: Distribution of correctly/wrongly released Audio resource over the app Categories

The compared results as shown in Figure 5.30 shows that:

- The percentage of correctly released on onPause() callback method for the *Audio* is 20% at *Multimedia* category, whereas the wrongly percentage is 7%.
- The percentages of wrongly released for the *Audio* are 17% at *Games*, 13% at *Sports & Health* and 7% at *Time* categories, whereas there are no correctly released.
- In the remaining categories, there are no occurrence for released *Audio* resource.
- The total average of the correctly released percentages on onPause() callback method for the *Audio* in all categories is 20%, whereas the total of the wrongly released percentages is 43%.

5.2.2.4.8 Network System's Resource *Network* system resource provide framework callled "Network.Net" that helps with network access [49]. This framework offers APIs such as ConnectivityManager "android.net.ConnectivityManager" that shows network connectivity state. Also, when connectivity changes, ConnectivityManager API notifies apps. Moreover, it Monitors network connections and sends broadcast intents. ConnectivityManager API

uses registerDefaultNetworkCallbac() , requestNetwork() and registerNetworkCallback() methods to acquire connection with Network resource. Also, it uses getSystemService(), unregisterNetworkCallback() ,removeDefaultNetworkActiveListener() ,releaseNetworkRequest(), or release() to close connection. Developer also can use NetworkRequest API to acquire/released connection with network resource.

(1) Acquiring Network system's resource:

Android apps which use an instance of NetworkConectivity resource to create the network object when the app starts using it [49]. Apps must acquire/open the *Network* resource inside onCreate() callback method. For example, in case a developer uses an instance of the ConnectivityManager API, the ConnectivityManager.getSystemService() method will be used as shown in the code bellow:

```
public class NetworkActivity extends Activity {
   private ConnectivityManager mConnectivityManager;
   @Override
   protected void onCreate() {
       super.onCreate()
      mConnectivityManager=(ConnectivityManager)getSystemService(Context.CONNECTIVI TY_SERVICE);
   }
}
```

After analyzing 5577 Android activities, the acquiring *Network* system resource percentages result were obtained as shown in the following Table 5.33. Table 5.33 shows the distribution of the acquired *Network* across the app categories. Each value represents the count and percentage of occurrence of the acquired *Network* over 13 existence of *Network* system resource inside 5577 activities. It shows that there are no occurrence for acquired *Network* resource in the all app categories.

TABLE 5.33: Distribution of acquired Network system's resource

Category	onCreate()	onStart()	onResume()	onPause()	onStop()	onDestroy()	onRestart()	OTHER
Connectivity	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Development	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Games	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Graphics	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Internet	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Money	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Multimedia	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Navigation	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Phone & SMS	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Reading	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Science & Education	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Security	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Sports & Health	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
System	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Theming	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Time	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Writing	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%

(2) Releasing Network system resource:

Android apps which used an instance of ConctvityManager resource must be particularly careful to release the network object when the app stops using it [49]. This occurs as soon as an app -activity- is paused , activity calls onPause() callback method. So, if Android apps does not properly release the network resource inside onPause() may cause these to be shut down. For example, in case a developer uses an instance of the ConectvityManager API, the ConectvityManager.release() method will be used as shown in the code bellow:

```
public class NetworkActivity  extends Activity  {
   private ConnectivityManager  mConnectivityManager;
   @Override
   protected void onPause() {
       super.onPause()
      mConnectivityManager.release();
   }
}
```

After analyzing 5577 Android activities the releasing *Network* system's resource percentages result were obtained as shown in the following Table 5.33. Table 5.33 shows the distribution of the released network API across the categories. Each value represents the count and percentage of occurrence of the released *Network* over 13 existence of *Network* system resource inside 5577 activities. It shows that there are no occurrence for released *Network* resource in the all app categories.

TABLE 5.34: Distribution of released Network system's resource

Category	onCreate()	onStart()	onResume()	onPause()	onStop()	onDestroy()	onRestart()	OTHER
Connectivity	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Development	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Games	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Graphics	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Internet	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Money	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Multimedia	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Navigation	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Phone & SMS	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Reading	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Science & Education	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Security	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Sports & Health	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
System	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Theming	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Time	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Writing	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%

5.2.2.4.9 USB System's Resource *USB* resource provides support to communicate with *USB* hardware peripherals that are connected to Android devices [50]. Android offers USB framework called "android.hardware.usb" to let developers access to USB resource. Using UsbManager API "android.hardware.usb.UsbManager" from USB framework helps to access the state of the USB and to communicate with connected hardware peripherals. This API uses

openDevice() or openAccessory() methods to acquire connection with USB resource. Whereas, it uses release() method to released connction. Moreover, Using UsbDevice API
"android.hardware.usb.UsbDevice" from USB framework to communicate with the hardware peripheral. This API uses openDevice() or releaseInterface() to acquire or release *Network* resource.

(1) Acquiring USB system's recourse:

Android apps uses an instance of UsbManager resource to create the USB object when the app starts using it [50]. Apps must acquire/open the *Network* resource inside onResume() callback method. For example, in case a developer uses an instance of the UsbManager API, the
UsbManager.openDevice() method will be used as shown in the code bellow:

```
public class USBActivity extends Activity  {
   private USBManager mUSBManager;
   @Override
   protected void onResume() {
       super.onResume();
       mUSBManager = manager.openDevice(device);
   }
}
```

After analyzing 5577 Android activities, the acquiring *USB* system resource percentages result were obtained as shown in the following Table 5.35. Table 5.35 shows the distribution of the acquired *USB* across the app categories. Each value represents the count and percentage of occurrence of the acquired *Network* over 19 existence of *USB* system resource inside 5577 activities. It shows that there are no occurrence for acquired *USB* resource in the all app categories.

TABLE 5.35: Distribution of acquired USB system's resource

Category	onCreate()	onStart()	onResume()	onPause()	onStop()	onDestroy()	onRestart()	OTHER
Connectivity	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Development	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Games	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Graphics	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Internet	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Money	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Multimedia	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Navigation	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Phone & SMS	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Reading	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Science & Education	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Security	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Sports & Health	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
System	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Theming	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Time	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Writing	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%

(2) Releasing USB system's resource:

Android apps which used an instance of USBManager resource must be particularly careful to release the USB object when the app stops using it [50]. This occurs as soon as an app -activity- is paused , activity calls onPause() callback method. For example, in case a developer uses an instance of the USByManager, the USByManager.release() method will be used as shown in the code bellow:

```
public class USBActivity extends Activity  {
   private USBManager mUSBManager;
   @Override
   protected void onPause() {
       super.onPause();
       mUSBManager.release();
   }
}
```

After analyzing 5577 Android activities, the releasing *USB* system resource percentages result were obtained as shown in the following Table 5.35. Table 5.35 shows the distribution of the released *USB* across the app categories. Each value represents the count and percentage of occurrence of the released *USB* over 19 existence of *USB* system's resource inside 5577 activities. It shows that there are no occurrence for released *USB* resource in the all app categories.

TABLE 5.36: Distribution of released USB system's resource

Category	onCreate()	onStart()	onResume()	onPause()	onStop()	onDestroy()	onRestart()	OTHER
Connectivity	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Development	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Games	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Graphics	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Internet	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Money	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Multimedia	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Navigation	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Phone & SMS	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Reading	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Science & Education	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Security	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Sports & Health	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
System	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Theming	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Time	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%
Writing	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%	(0) 0%

5.3 Nature of Code Implemented Inside Callback Methods

The third part of our analysis focuses on the nature of code inside the most important callback methods which are the onPause(), onStop() and onDestroy(). This helps to decide a heavy and long running code. However, in order to decode the long running code, the suitable method is to compute the execution time for each method. But, this

might be very hard due to that our study use a static code analysis. What's more, the execution time has been influenced by the different mobile device fragmentation.

To solve this, the nature of code was considered according to main three actions. So that, the nature of code was divided into three categories. The first is releasing actions; the second is database actions; the third is threading actions. However, the second and third categories were considered (database and threading actions) as a long running and heavy code actions according to Android documentation site[17]. Each action was defined using some of the main keyword that developers might use when they use these actions. Using SAALC, the analysis was applied depending on these keywords of the categories for the onPause(), onStop() and onDestroy().

For example, to decide the nature of code, three categories were decided as:

- Category I: When the developer deal with releasing actions, some keywords such as pause, close, release, remove keyword were used.
- Category II: When the developer deal with database actions, some keywords such as sqlite data base or shared preferences were used.
- Category III: When the developer deal with threading actions, some keywords such as runnable, thread and sleep keywords were used.

Note: In case of onPause() callback method, category I did not include in our analysis due to avoiding a conflict with the main usage of the nPause() callback method to close and release connection, but the second and third categories(long running code) were used only

The result of this part shown in Table 5.37.

TABLE 5.37: Nature of code analysis

#	Category Name	onPause()	onStop()	onDestroy()
I	Releasing Resources actions	-	(50) 15%	(214) 27%
II	Database actions	(104) 12%	(35) 10%	(41) 5%
III	Threading actions	(43) 5%	(8) 2%	(45) 6%
Total		(147) 17%	(341) 27%	(780) 38%

Table 5.37 shows the percentages of the nature of code inside the onPause(), onStop() and onDestroy(). The results were also represented in Figure 5.31.

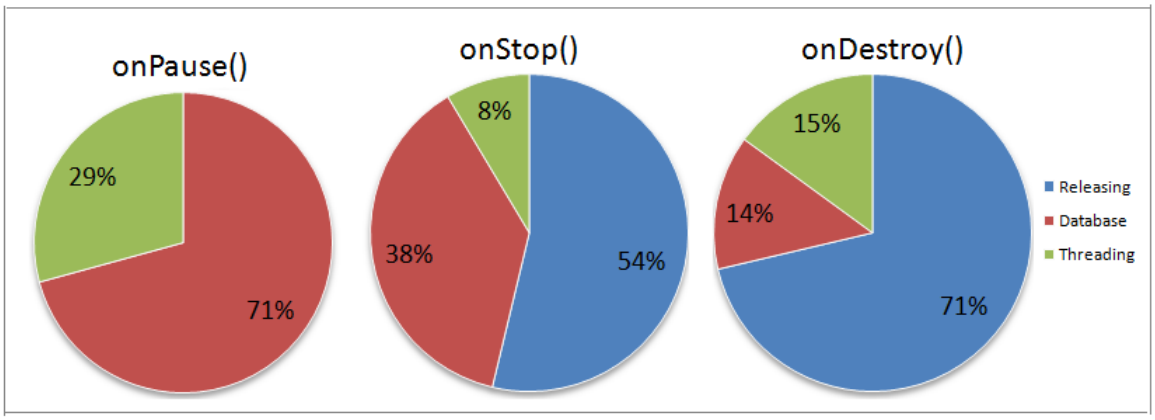


FIGURE 5.31: Nature of analysis

Figure 5.31 shows that:

- The total percentages for the nature of code inside inside the onPause() is (147) 17%. This includes the percentages of long running code (104) 12% for database actions; and (43) 5% for threading actions.
- The total percentages for the nature of code inside the onStop() is (341) 27%. This includes (50)15% for releasing actions; (43) 12% for long running code which includes (35) 10% for database actions; and (8) 2% for threading actions.

- The total of percentages for the nature of code inside onDestroy() is (780) 38%. This includes (214) 27% for releasing actions, (86) 11% for long running code which includes (41) 5% for database actions; and (45) 6% for threading actions.

In the next chapter the result of this these was discussed and the research questions were answered.

Chapter 6

Discussion

Development activity is not an easy process as it requires developers to have a better understanding of their applications and on how users will use them [3, 9]. Developers also need to be aware of the common lifecycle practice of available for mobile apps and correct the bad way of activity development. After analyzing several activities, researchers wanted to know whether Android developers correctly deal with the activity's lifecycle. So that, the set of research questions was answered to understand the activity lifecycle practices commonly followed by Android developers.

This section provided a discussion of the results in terms of the proposed research questions. It's divided into three subsections, each of them gives the resulted arguments and indications for each research question.

6.1 Utilizing Lifecycle Callback Methods

In the first part of our research, the usage of lifecycle in Android activities was considered to understand by analyzing whether developers use callback methods for their activities. This leads us to our first research question:

RQ1: To what extent Android developers utilize the lifecycle callback methods in developing mobile apps?.

Answering the RQ1 gives us the first indication about Android lifecycle callback methods' utilization. The results in section 5.1 show that onCreate() callback method is the one that is mostly implemented (92%). This is not surprising since onCreate() method is the main method to start and setup Android activities [17]. Implementing the onCreate() callback method is important to initialize the UI of the activity as well as other data binding operations. Moreover, developers use onCreate() method to do all normal create views and setup.

The onResume() callback method is implemented by (23%). It is normally called when the activity is in the foreground and about to start interacting with users' interactions[17]. Additionally, it is used for acquiring system resources among other services by developers. Thus, the second indication can be assumed that such percentage of utilizing the onResume() method is reasonable due to the limited number of activities that deal with managing system resources around 3% as shown in section 5.2 .

The onPause() callback method usage percentage is (16%) and is normally used when an activity is about to go to the background [17]. More specifically, it is used to commit unsaved changes, release system resources, stop animations and other processes that can consume the CPU. This result can be concluded here that such usage percentage reasonable due to the same reasons as shown in the onResume().

On the other hand, the onRestart() callback method is rarely implemented (1%). The onRestart() callback method is normally used after an activity has stopped and before it is started again [17]. Additionally, it is used to acquire a raw cursor objects if developers have already deactivated it at onStop() method [17]. Accordingly, This gives us the third indication, it can be assumed that the usage percentage of onRestart()

method (1%) is also reasonable since it has limited usage scenarios in case developers create a cursor implementation object and required it on onRestart() callback method instead they operate database query objects through managedQuery().

The onStart() and onStop() callback methods both have usage percentage of (6%). The onStart() callback method is normally called when an activity is newly created and becoming visible to the user [17]. However, Google documentation does not provide a clear description of when to use this method. Whereas, the onStop() callback method is normally used to save app data to permanent storage and to execute long running code[17]. According to [17], developers must execute long running code at onStop() instead of onPause() method. This is due to the fact that onPause() method must be executed quickly so that other activities can start seamlessly. From our perspective, 6% for onStart() and onStop() are reasonable percentages.

Additionally, the forth indication was reflected according to these two arguments. The first is that Android developers might less understandable of onStart() or onStop() method usage due to insufficient describing on Android documentation. And the second, that developers understand the usage of them, but they seem less important to use them. Both of these arguments give us the forth indication that Android documentation needs to be more useful and give developer complete detail on how to use Android callback methods.

The onDestroy() callback method has usage percentage of (14%). The onDestroy() callback methods normally called before an activity is destroyed [17]. It acts as the last chance for developers to free resources and threads that are associated with the activity before it is removed from memory [17]. However, according to the study by [8], the onStop() and onDestroy() methods may not be called by the system in cases where the system is very low in resources (battery and memory). In such situation, developers may face a dilemma. This is because both the onStop() and onDestroy()

methods are normally used to execute long running code. This will be had more details in the discussion RQ3.

6.2 Utilizing Android System's Resources

In second part of our research, the usage of system's resource was considered to understand by analyzing activities source code and collecting statistics about main system's resources such as *Camera and Bluetooth, GPS, sensors*, etc. Utilizing system's resource showed us whether Android developers acquired and released the system's resource correctly as compared to Google documentation. So that's lead to the second research question is:

RQ2: Did Android developer correctly acquire and release the Android system resources?.

The averages of correctly acquired and released of the previous nine resources are equal to (23%) and (11%) respectively. However, the averages of wrongly acquired and released resources of previous nine are (16%) and (8%) respectively. The result was concluded here that a large percentage of system resources are not correctly released by the developers. As a consequence, according to study [9], this will lead to incorrect behavior of the Android app, as well as memory leaks and runtime errors. This result gives us three arguments. Firstly, developers are not might fully aware of the importance of correctly managing the system resources. Secondly, developers possibly aware of the importance of correctly managing the system's resources, but they take it for granted. However, thirdly the main reference for developers to learn how to manage the system resources is unclear and it's hard to understand how doing that using Android documentation. So, the indication here nears to the previous indication in RQ1 that Android documentation needs to be more complete and

clear enough useful information about managing system resources, because this will influence the percentages of correctly/wrongly of acquired and released system's resources.

6.3 Utilizing Nature of Code Implemented inside Callback Methods

In the third part of our research, the nature of code inside callback methods was analyzed. All activities were analyzed to collect what kind of code was implemented inside onPause(), onStop() and onDestroy() callback methods. So, this question will give us some statistics about how the developer uses these callback methods. And, if they implemented a long running code or not.

RQ3: What is the nature of code implemented inside onPause(), onStop, and onDestroy() callback methods?.

Regarding the nature of code implemented inside the onPause(), onStop() and onDestroy(), the result shows that the onStop() method has a (27%) nature of code includes (12%)that is considered to be long running code. Further the onDestroy() method has a (38%) includes (11%) of long running code. This is acceptable from the point of view of the Android official documentation [17]. However, the problem here is that these two methods may not be called according to the study by [8].

An the same time the developer cannot write a long running code inside the onPause(). This is due to the fact that the onPause() should be executed quickly so that other activities can be started [17]. Regarding the nature of code inside onPause() call back method, it has a (17%) of code that is considered to be long running. This is considered to be an issue as discussed above, since this will possibly block other

activities from running seamlessly.

In the next Chapter, the conclusion of our thesis was presented. Moreover, some threats to validity, obstacles and future work were introduced.

Chapter 7

Conclusions

7.1 Conclusion

Android activity lifecycle model is very important to understand in order to develop robust apps. With an ever-growing app community, activity lifecycle holds more importance to ensure that apps are adequately reliable and robust. Our study is the first study to explore the usage of activity's lifecycle callback methods in the Android development community. A tool called SAALC was built to analyze 5577 activities residing in 842 Android apps from F-Droid repository. The activities were analyzed to collect statistics about the utilization of each callback method; the averages of correct and wrong acquired/released system resources; and the nature of long running process inside onPause(), onStop() and onDestroy() callback methods. Our findings can be summarized as follows:

- The occurrence percentages of callback methods are about 1% of the activities used onRestart() method, 6% used onStop(),23% used onResume(), 16% used onPause() and 14% used onDestroy(). The most occurrences for onCreate() callback methods about 92% of activities.

- Only about 3% of the activities contain Camera, Audio, Bluetooth, Database, GPS, Input, Network, Sensor and USB system's resources.
- The wrongly acquired average of system's resource is 16%, whereas the wrongly released average is 8%. That's will influence the app reliability.
- About 17% of activities used long running code inside onPause() callback methods and that will influence the app performance.
- About 27% of activities used releasing and long running code actions inside onStop()callback method, whereas 38% used inside onDestroy() and there is no guarantee to be called inside lifecycle.

These findings show that Android developers, in general, have limited knowledge and awareness of the importance of writing an app that conforms to the lifecycle model. Thus, this will affect apps' reliability and performance. Further, the result was argued that Android documentation needs to be more useful, complete and clear in describing how developers will use activity callback method and system's resources. Android developers can use our findings to gain insights into Android apps development. Moreover, software researchers can use our findings to provide support for developers by extending more research in this fields, in order to help developers of building more robust apps.

7.2 Difficulties and Obstacles

Due to that, this thesis is the first exploration, study about lifecycle conformance. It was hard to gain proved result using reasonable benchmarks from previous studies. However, the arguments and indications which founded in the discussion chapter due to some facts related to lifecycle model and Android documentation.

7.3 Threat to validity

SAALC was developed to relate with most patterns and style of coding which developers may depend. However, Threats to internal validity considered the situations and conditions under which experiments are implemented. These are some of the Threats to internal validity was founded:

- Apps which contain system's resource was automatically identified using import package in .java file. Sometimes, activity files that used resources missed due to the automatic process to search for the package names.
- The Java parser component was used from an online source without ensuring its efficacy or testing it. However, it was used and tested before by Zein et al. study [3].
- Fields name was decided according to the type of resources from the field's list inside .java file to check where the resource acquired and released inside the file methods. However, Sometimes developers insert useless resource which declared as a field and did not call any more. This will affect the count of the occurrences results in our analysis.
- Sometimes the inheritance and casting characteristics of JAVA language allow a developer to implement levels of sub child or different coding styles to acquire or release resources. So, in these cases, it's hard to decide if the resource was acquired or released depending on the field type declaration.
- Sometimes developers released or acquired a resource more than one time in the same methods. This condition was resolved by considering the frequency of occurrence for acquired or released statement inside a method for one time and did not compute all statements were found.

- Some of Internal threats occur when trying to decide a long ruining code inside callback methods. A time execution of a static code was not calculated due to that effect by the difference between mobiles fragmentation and hardware. Instead, it was decided to use some actions that influenced a performance such as using a database and threading.

SAALC was used to analyze over than 5000 Android activities. However, aware was taken to threats to external validity, which considered to the generalizability of our results. There are some of the Threats to external validity was founded due to the reasons of over than 5000 Android activities was investigated from F-Droid, which is one of the largest repositories of open source Android apps. Our dataset included of many kinds of activities that's had small ones to large from different categories. However, it's insufficient to if ours patterns findings would generalize to all Android activities from different dataset. To overcome the external validity, in the future, more activities from other datasets will be used.

7.4 Future Work

Our study is the first exploratory step to understand the activity lifecycle practices in the app development community. In the future, our study will be expanded by analyzing more apps from different platforms such as IOS and by adding mining and analyzing techniques to our approach. Also, suitable benchmarks to measure the correctness of lifecycle usage will be studied to prove our results and provide meaningful indications about the best lifecycle conformance practices.

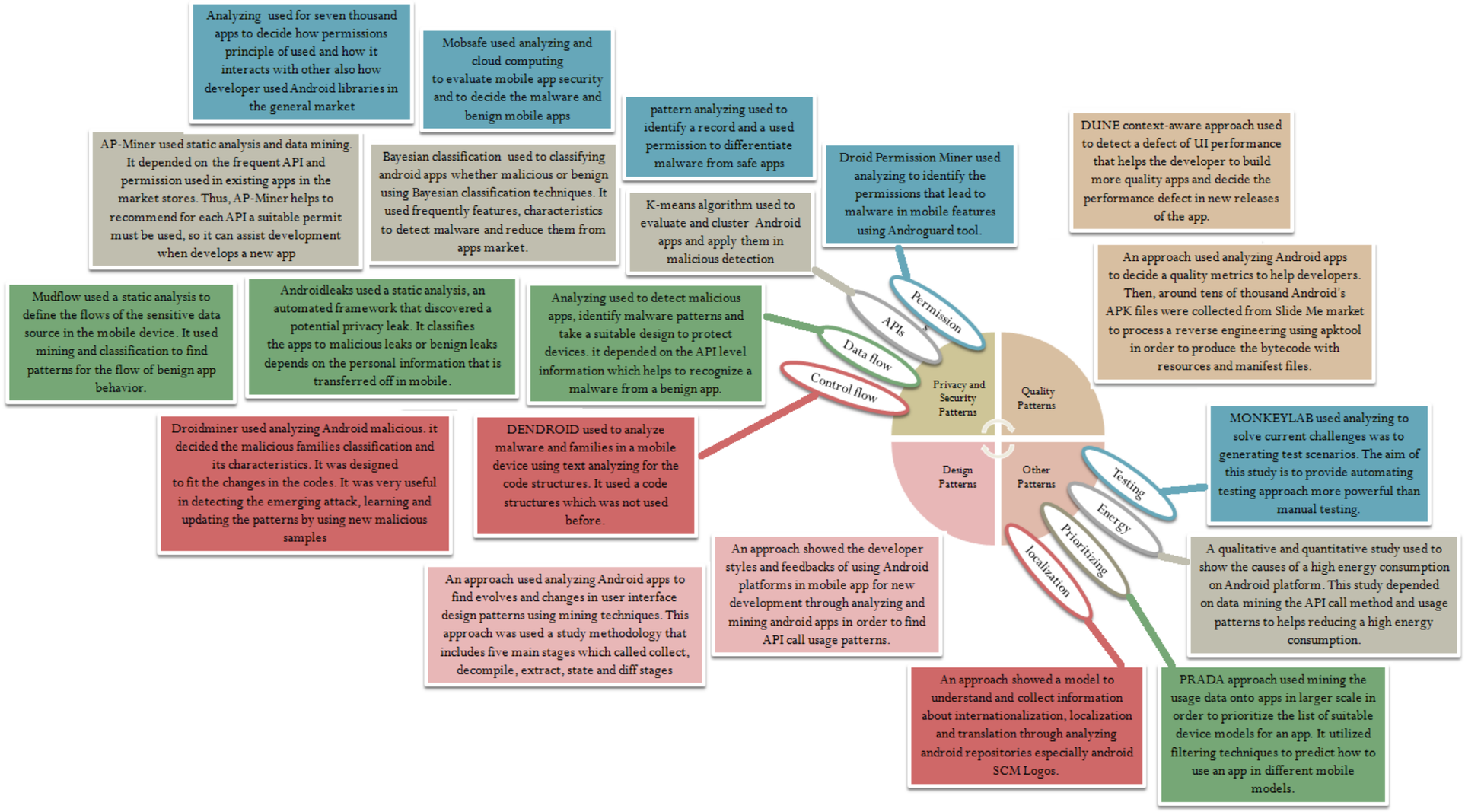
Appendix A

A.1 Repository information

TABLE A.1: Resources repository information from Android documentation [17]

Resources	Package name	Name of acquiring method	Name of releasing method	Name of acquired callback	Name of released callback
Camera	Camera	open()	release()	onResume()	onPause()
		startPreview()	stopPreview()		
	Camera2	open()	release()		
		startPreview()	stopPreview()		
	CameraManager	openCamera()	release()		
	CameraDevice	openCamera()	close()		
		onOpened()	onClosed		
USB	UsbManager	openDevice()	release()	onCreate()	onPause()
		openAccessory()			
	UsbDeviceConnection	openDevice()	releaseInterface()		
Sensor	SensorManager	registerListener()	unregisterListener()	onResume()	onPause()
		start()	stop()		
		getsystemservice()			
Network	Network	openConnection()	release()	onCreate()	onPause()
		registerDefaultNetworkCallback()	unregisterNetworkCallback()		
		requestNetwork()	releaseNetworkRequest()		
	ConnectivityManager	registerNetworkCallback()	removeDefaultNetworkActiveListener()		
	NetworkRequest	registerNetworkCallback()	release()		
Input	InputManager	registerInputDeviceListener()	unregisterInputDeviceListener()	onCreate()	onPause()
		getsystemservice()	stop()		
		start()			
GPS	LocationManager	getGpsStatus()	stop()	onCreate()	onPause()
			release()		
		start()	cancel()		
DataBase	SQLiteDatabase	openOrCreateDatabase()	close()	onCreate()	onPause()
	Context	openOrCreateDatabase()	close()		
	SQLiteClosable	openDatabase()	releaseMemory()		
			close()		
			releaseReference()		
			releaseReferenceFromContainer()		
Bluetooth	BluetoothAdapter	getDefaultAdapter()	stopLeScan()	onCreate()	onPause()
		enable()	cancelDiscovery()		
		startLeScan(0	closeProfileProxy()		
		startDiscovery()			
		getDefaultAdapter()			
		getbondeddevices()			
	BluetoothDevice()	accept()	close()		
	BluetoothServerSocket()	accept()	close()		
	BluetoothSocket()	accept()	close()		
			cancel()		
			release()		
Audio	AudioRecord	startRecording()	stop()	onCreate()	onPause()
	AudioManager	startRecording()	getsystemservice()		

A.2 Literature review studies



A.3 SAALC Class Diagram

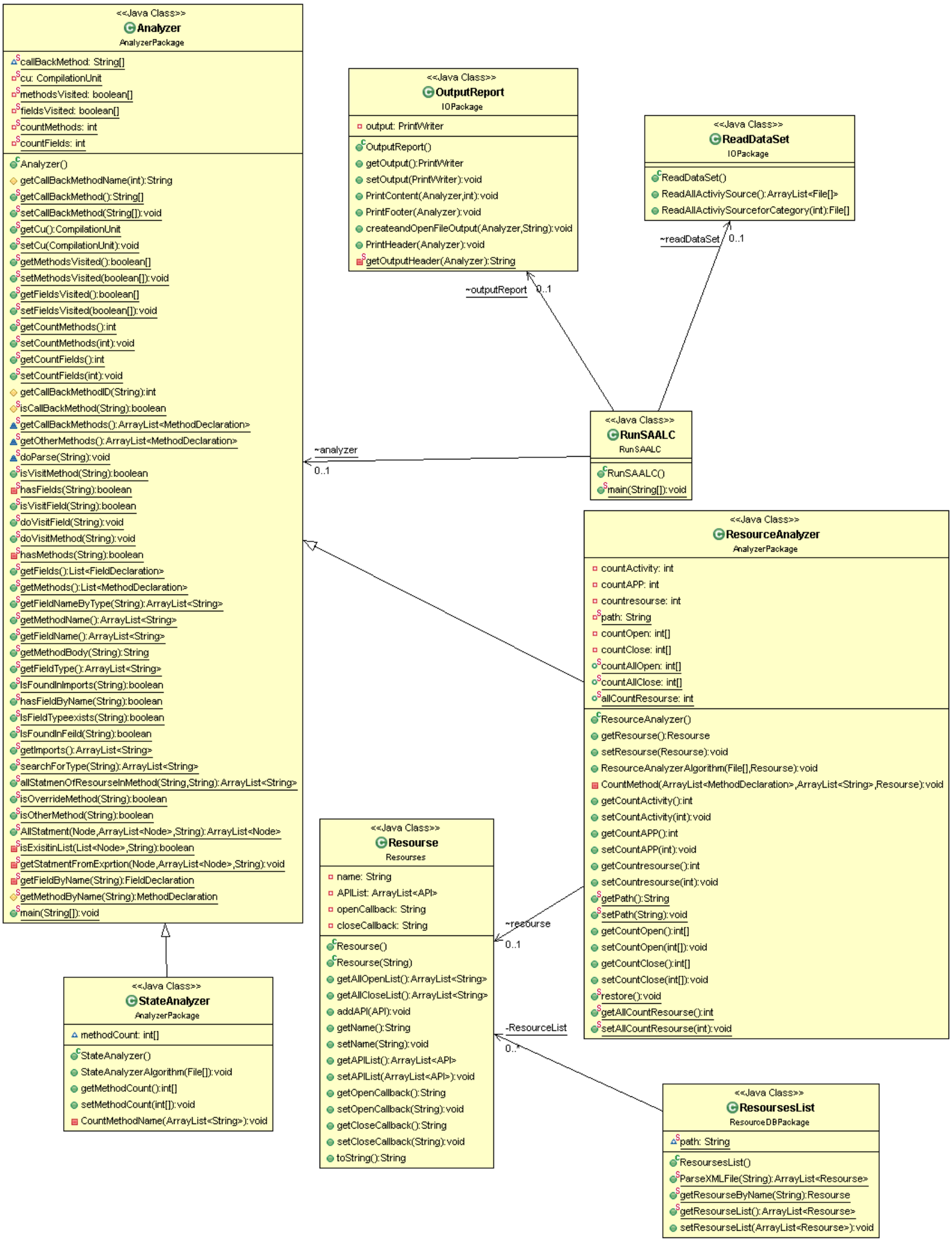


FIGURE A.1: The class diagram for SAALC tool

References

[1] Anthony I Wasserman. "Software engineering issues for mobile application development". In: *Proceedings of the FSE/SDP workshop on Future of software engineering research*. ACM. 2010, pp. 397–400.

[2] Josh Dehlinger and Jeremy Dixon. "Mobile application software engineering: Challenges and research directions". In: *Workshop on Mobile Software Engineering*. Vol. 2. 2011, pp. 29–32.

[3] Samer Zein et al. "Static analysis of android apps for lifecycle conformance". In: *Information Technology (ICIT), 2017 8th International Conference on*. IEEE. 2017, pp. 102–109.

[4] Yash Lamba et al. "Pravaaha: Mining Android applications for discovering API call usage patterns and trends". In: *Proceedings of the 8th India Software Engineering Conference*. ACM. 2015, pp. 10–19.

[5] *App stores: number of apps in leading app stores 2017*. 2017-05-21 15:38:32. URL: `https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/m` (visited on 2017).

[6] Samer Zein, Norsaremah Salleh, and John Grundy. "A systematic mapping study of mobile application testing techniques". In: *Journal of Systems and Software* 117 (2016), pp. 334–356.

[7] Mona Erfani Joorabchi, Ali Mesbah, and Philippe Kruchten. "Real challenges in mobile app development". In: *Empirical Software Engineering and Measurement, 2013 ACM/IEEE International Symposium on*. IEEE. 2013, pp. 15–24.

[8] Dominik Franke et al. "Reverse engineering of mobile application lifecycles". In: *Reverse Engineering (WCRE), 2011 18th Working Conference on*. IEEE. 2011, pp. 283–292.

[9] Dominik Franke et al. "Testing conformance of life cycle dependent properties of mobile applications". In: *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*. IEEE. 2012, pp. 241–250.

[10] Sebastiano Panichella et al. "Would static analysis tools help developers with code reviews?" In: *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*. IEEE. 2015, pp. 161–170.

[11] Zhang Haotian and Liu Shu. "Java Source Code Static Check Eclipse Plug-In Based on Common Design Pattern". In: *Software Engineering (WCSE), 2013 Fourth World Congress on*. IEEE. 2013, pp. 165–170.

[12] Phongphan Danphitsanuphan and Thanitta Suwantada. "Code smell detecting tool and code smell-structure bug relationship". In: *Engineering and Technology (S-CET), 2012 Spring Congress on*. IEEE. 2012, pp. 1–5.

[13] Shaheen Khatoon, Azhar Mahmood, and Guohui Li. "An evaluation of source code mining techniques". In: *Fuzzy Systems and Knowledge Discovery (FSKD), 2011 Eighth International Conference on*. Vol. 3. IEEE. 2011, pp. 1929–1933.

[14] Alexander Ramos. *Evaluating the ability of static code analysis tools to detect injection vulnerabilities*. 2016.

[15] Veelasha Moonsamy et al. "Mining permission patterns for contrasting clean and malicious android applications". In: *Future Generation Computer Systems* 36 (2014), pp. 122–132.

[16] Suleiman Y Yerima et al. "A new android malware detection approach using bayesian classification". In: *Advanced Information Networking and Applications (AINA), 2013 IEEE 27th International Conference on*. IEEE. 2013, pp. 121–128.

[17] *Activity | Android Developers*. URL: `https://developer.android.com/reference/android/app/Activity.html` (visited on 11/01/2017).

[18] Jill Jesson and Fiona Lacey. "How to do (or not to do) a critical literature review". In: *Pharmacy education* 6 (2006).

[19] *Data Mining vs. Statistics - How Are They Different?* Dec. 2015. URL: `https://www.simplilearn.com/data-mining-vs-statistics-article` (visited on 09/10/2017).

[20] Aswini et al. "Droid permission miner: Mining prominent permissions for Android malware analysis". In: *Applications of Digital Information and Web Technologies (ICADIWT), 2014 Fifth International Conference on the*. IEEE. 2014, pp. 81–86.

[21] Yousra Aafer et al. "Droidapiminer: Mining api-level features for robust malware detection in android". In: *International Conference on Security and Privacy in Communication Systems*. Springer. 2013, pp. 86–103.

[22] Md Yasser Karim et al. "Mining android apps to recommend permissions". In: *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*. Vol. 1. IEEE. 2016, pp. 427–437.

[23] Jianlin Xu et al. "MobSafe: cloud computing based forensic analysis for massive mobile applications using data mining". In: *Tsinghua Science and Technology* 18.4 (2013), pp. 418–427.

[24] Matthew L Dering et al. “Android market reconstruction and analysis”. In: *Military Communications Conference (MILCOM), 2014 IEEE*. IEEE. 2014, pp. 300–305.

[25] Aiman A Abu Samra et al. “Analysis of clustering technique in android malware detection”. In: *Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS), 2013 Seventh International Conference on*. IEEE. 2013, pp. 729–733.

[26] Clint Gibler et al. “AndroidLeaks: automatically detecting potential privacy leaks in android applications on a large scale”. In: *International Conference on Trust and Trustworthy Computing*. Springer. 2012, pp. 291–307.

[27] Vitalii Avdiienko et al. “Mining apps for abnormal usage of sensitive data”. In: *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press. 2015, pp. 426–436.

[28] Guillermo Suarez-Tangil et al. “Dendroid: A text mining approach to analyzing and classifying code structures in android malware families”. In: *Expert Systems with Applications* 41.4 (2014), pp. 1104–1117.

[29] Chao Yang et al. “Droidminer: Automated mining and characterization of fine-grained malicious behaviors in android applications”. In: *European Symposium on Research in Computer Security*. Springer. 2014, pp. 163–182.

[30] Khalid Alharbi and Tom Yeh. “Collect, decompile, extract, stats, and diff: Mining design pattern changes in Android apps”. In: *Proceedings of the 17th International Conference on Human-Computer Interaction with Mobile Devices and Services*. ACM. 2015, pp. 515–524.

[31] Eric Shaw et al. “Mining Android apps to predict market ratings”. In: *Mobile Computing, Applications and Services (MobiCASE), 2014 6th International Conference on*. IEEE. 2014, pp. 166–167.

[32] María Gómez et al. “Mining test repositories for automatic detection of UI performance regressions in Android apps”. In: *Proceedings of the 13th International Conference on Mining Software Repositories*. ACM. 2016, pp. 13–24.

[33] Mario Linares-Vásquez et al. “Mining android app usages for generating actionable gui-based execution scenarios”. In: *Proceedings of the 12th Working Conference on Mining Software Repositories*. IEEE Press. 2015, pp. 111–122.

[34] Mario Linares-Vásquez et al. “Mining energy-greedy api usage patterns in android apps: an empirical study”. In: *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM. 2014, pp. 2–11.

[35] Xuan Lu et al. “PRADA: Prioritizing android devices for apps by mining large-scale usage data”. In: *Proceedings of the 38th International Conference on Software Engineering*. ACM. 2016, pp. 3–13.

[36] *F-Droid - Free and Open Source Android App Repository*. URL: https://f-droid.org/ (visited on 11/05/2017).
[37] Daniele Simonin. *FOSS applications statistics*. Dec. 2016. URL: https://fossdroid.com/blog/foss-applications-statistics.html (visited on 05/30/2017).
[38] *A test automation tool*. URL: http://sahipro.com/ (visited on 11/01/2017).
[39] *Java parser*. URL: https://github.com/javaparser/javaparser (visited on 11/01/2017).
[40] *Camera API | Android Developers*. URL: https://developer.android.com/guide/topics/media/camera.html (visited on 11/26/2017).
[41] *android.database.sqlite | Android Developers*. URL: https://developer.android.com/reference/android/database/sqlite/package-summary.html (visited on 11/26/2017).
[42] *Database mangement and the Activity lifecycle*. Mar. 2010. URL: https://awiden.wordpress.com/2010/03/26/database-mangement-and-the-activity-lifecycle/ (visited on 11/26/2017).
[43] *Sensors Overview | Android Developers*. URL: https://developer.android.com/guide/topics/sensors/sensors_overview.html (visited on 11/27/2017).
[44] *Making Your App Location-Aware | Android Developers*. URL: https://developer.android.com/training/location/index.html (visited on 11/27/2017).
[45] *LocationManager | Android Developers*. URL: https://developer.android.com/reference/android/location/LocationManager.html (visited on 11/26/2017).
[46] *InputManager | Android Developers*. URL: https://developer.android.com/reference/android/hardware/input/InputManager.html (visited on 11/26/2017).
[47] *BluetoothAdapter | Android Developers*. URL: https://developer.android.com/reference/android/bluetooth/BluetoothAdapter.html (visited on 11/27/2017).
[48] *AudioManager | Android Developers*. URL: https://developer.android.com/reference/android/media/AudioManager.html (visited on 11/27/2017).
[49] *ConnectivityManager | Android Developers*. URL: https://developer.android.com/reference/android/net/ConnectivityManager.html (visited on 11/27/2017).
[50] *android.hardware.usb | Android Developers*. URL: https://developer.android.com/reference/android/hardware/usb/package-summary.html (visited on 11/27/2017).