MEASURING AND ASSESMENT OF WELL KNOWN BAD PRACTICES IN ANDROID APPLICATION DEVELOPMENTS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF INFORMATICS
OF
THE MIDDLE EAST TECHNICAL UNIVERSITY

BY

İSMAİL ALPER SAĞLAM

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE
DEGREE OF
MASTER OF SCIENCE
IN THE DEPARTMENT OF INFORMATION SYSTEMS

SEPTEMBER 2014

MEASURING AND ASSESMENT OF WELL KNOWN BAD PRACTICES IN ANDROID APPLICATION DEVELOPMENTS

Submitted by **İSMAİL ALPER SAĞLAM** in partial fulfillment of the requirements for the degree of **Master of Science in Information Systems, Middle East Technical University** by,

Prof. Dr. Nazife Baykal ______________________
Director, **Informatics Institute**

Prof. Dr. Yasemin Yardımcı Çetin ______________________
Head of Department, **Information Systems**

Assoc. Prof. Dr. Aysu Betin Can ______________________
Supervisor, **Information Systems, METU**

Examining Committee Members:

Prof. Dr. Kürşat Çağıltay ______________________
CEIT, METU

Assoc. Prof. Dr. Aysu Betin Can ______________________
IS, METU

Dr. Ali Arifoğlu ______________________
IS, METU

Assoc. Prof. Dr. Erhan Eren ______________________
IS, METU

Assoc. Prof. Dr. Banu Günel ______________________
IS, METU

Date: 15.09.2014

ABSTRACT

MEASURING AND ASSESMENT OF WELL KNOWN BAD PRACTICES IN ANDROID APPLICATIONS

SAĞLAM, İSMAİL ALPER

M.Sc., Department of Information Systems

Supervisor: Assoc. Prof. Dr. AYSU BETİN CAN

September 2014, 67 Pages

One of the best ways to make a mobile application usable, reputed and high-scored is attention to the requirements like responsiveness, low memory consumption and stability. To meet these requirements developers must improve their codes by avoiding some bad-practices, which cause "Memory-Leaks", "ANR (Application not responding)" and "Out-of-Memory" to satisfy the user's need and make the Android application robust and usable. In this thesis, I developed a tool that detects a set of bad-practices in Android applications automatically. The tool is applied to source code of 100 open source Android applications. The findings of the tool are used to analyze whether there is a relationship between the user ratings (i.e. the reputation) of the applications with the number and type of bad-practices. To represent reputation, the statistical data of the 100 Android applications that shows their success such as rating and install count is collected from the applications' official web sites. Another contribution is that, with the aid of the tool developed in this study, developers will be able to find their mistakes in their codes easily or know what may go out wrong when they release their Android applications.

Keywords: android, bad-practices, memory-leak, ANR, automation

ÖZ

ANDROİD UYGULAMASI GELİŞTİRİLMELERİNDE YAPILAN YANLIŞ YÖNTEMLERİN ÖLÇÜMÜ VE DEĞERLENDİRİLMESİ

SAĞLAM, İSMAİL ALPER

Yüksek Lisans, Bilişim Sistemleri Bölümü

Tez Yöneticisi: Doç. Dr. AYSU BETİN CAN

Eylül 2014, 67 sayfa

Bir mobil uygulamayı daha kullanışlı, tanınmış ve yüksek skorlu yapmanın en iyi yollarından biri bu uygulamayı daha cevap verebilir, az hafıza tüketen ve kararlı olma gereksinimlerini yerine getirmektir. Bu gereksinimleri yerine getirebilmek için geliştiriciler uygulamalarını "Bellek-Sızıntısı", "ANR (Uygulama Yanıt Vermiyor)" ve "Yetersiz-Bellek" hatalarına sebep olan bazı kötü yöntemlerden ayırıp kullanıcıların ihtiyaçlarına cevap vermelidirler. Bu çalışmada Android uygulamalarını inceleyen ve bu uygulamalardaki kötü yöntemlerin ortaya çıkarılmasını otomatik hale getiren bir araç geliştirilmiştir. Bu araç 100 açık kaynak kodlu uygulama üzerinde çalıştırılmıştır. Aracın bulduğu sonuçlar ile uygulamaların kullanıcı derecelendirmeleri arasındaki ilişki incelenmiştir. Kullanıcı değerlendirme verisi olarak uygulamaların resmi web sitelerinden kullanıcı derecelendirmeleri, indirilme sayıları bilgisi toplanmıştır Bunların yanı sıra, geliştirilen araç sayesinde geliştiriciler kodlarındaki hataları daha kolay bulabilecek ve uygulamayı piyasaya sürdüklerinde nelerin yanlış olabileceğini anlayacaklardır.

Anahtar Kelimeler: android, kötü-yöntemler, bellek-sızıntısı, ANR, otomasyon

ACKNOWLEDGEMENTS

I express my sincere appreciation to my supervisor, Assoc. Prof. Dr. Aysu Betin Can, for her guidance, insight, support, encouragement and positive attitudes throughout the study. I am thankful for having the chance of working with her.

I would like to express my very great appreciation to Hande Taşlıoğlu, for his precious support, his endless love and for being the source of my motivation. It would be impossible to finish this work without his support.

I am very grateful and would like to thank my family for their invaluable patience and encouragement.

I would like to thank all of my friends for their support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

ANR	Application Not Responding
ADT	Android Development Tools
AST	Abstract Syntax Tree
DVM	Dalvik Virtual Machine
FOSS	Free and Open Source Software
SDK	Software Development Kit
UI	User Interface

CHAPTER 1

INTRODUCTION

Creating a top-selling, stable and high-rated mobile application is not an easy task to do. Many mobile application developers and companies especially the ones working on Android must have an understanding of mobile applications have a huge difference from traditional ones because the resources are limited and the platform is eager to hang the developers out the dry. To have a good reputation, mobile application developers must obey some special rules that are governed by the environment of mobile application such as users must have ability to personalize everything [1] and they should always be careful about not having a bad reputation by annoying users.

The desired qualities for a mobile application according to [2] are**:**

- **More Stable**:

  Application must not crash, force close, freeze or function unnatural on any targeted device and get rid of possible bugs.

- **More Responsive**:

  Application must not do potentially long operations on user interface (UI) thread and no *Application Not Responding* (ANR) dialog will be displayed.

  100 ms to 200 ms is the key period for users not to detect any slowness in an application [3]. Other than that if there are long tasks, instead of freezing, application must give a hint about the progress of the current process.. In Android Operating System, there is a guarding system that is against the applications that are sluggish, hanging or freezing for significant period of time or that cannot take any new inputs from the user in too long periods. In these situations, system offers the user to exit from the application. It is the responsibility of developers to design the application that never shows this offer to user by preventing such situations. [4]**.** The conditions that activate the guarding system are as follows:

  - In 5 seconds, there is no response from the application to an input event.
  - In 10 seconds, an execution that caused from a BroadcastReceiver (which is discussed in Chapter 2) is not finished.

- **Quicker:**

  Application must load rapidly and eliminate unnecessary actions or features because speed matters**.** According to experiment by Google researchers [3], if the search time delay is intentionally increased to 100 ms to 400 ms, users begin to search 0.2% to 0.6% less often. Moreover, even if this intentional delay is withdrawn, the search rate of users will never be increased to their first level. Since these numbers are for Google, they may seem small but if the speed matters for even that big software, it is expected more effective for any standalone application that in Android world.

In mobile application world, there is always a market that operates like clockwork such as Google Play[1] . These markets make developers easier to earn a reputation for their product and even sell them. Applications with good reviews and high ratings tend to get better revenue, so good reputation is an important factor. There are several books [5] [6] and off-line [7], on-line [8] courses that teach Android application development frameworks and how to build applications with good quality. Beside, some blogs, tutorials and articles focus on the common pitfalls in the Android application framework and they discuss choosing the right way of doing a task among many ways that end up same result.

In this thesis, we collected common mistakes that decrease these qualities in an Android application. After that, we converted those mistakes into patterns that can be recognized in Android application source code. This conversion was followed by developing a tool that detects those patterns in an Android project automatically. This tool is the first contribution of this thesis.

In parallel, we downloaded source codes of open source applications and collected their review statistics, i.e. rate counts for each point 1 to 5, average rate and install count data of those open source applications from *Google Play*. This statistic was used as a raw material for calculating success of applications. The tool developed to detect the bad-practices was run with all of the projects that are downloaded. The output of the tool was used to produce a dataset, which includes how many of those applications contain bad-practices, how many bad-practice each application project contains and bad-practice distribution in successful, unsuccessful projects.

This thesis also aims to answer to the question “Does having a set of chosen bad-practices in its source code cause lower ratings for an application?”. The extracted statistical data is summed up to answer to this question.

The rest of the thesis is organized as follows: Chapter 1 introduces our work and gives the idea behind it, Chapter 2 gives the background information that is needed for understand the rest of the thesis, Chapter 3 shows the other works in this field and their difference from our work, Chapter 4 explains the methodology for how our work is put

[1] https://play.google.com/store

into practice, Chapter 5 presents analysis of this work and Chapter 6 is the conclusion of this thesis that clarifies the findings, limitations and future work.

CHAPTER 2

BACKGROUND INFORMATION

2.1. What is Android?

Android is an operating system designed for mobile devices like smartphones or tablets with touchscreens. This operating system is based on Linux Kernel handling low-level hardware interactions providing set of APIs to access underlying services and features [9]. [2]

2.2. Android User Interface

As default, Android Operating System has a direct manipulation user interface [9], which is a human computer interaction style involving "continuous representation of objects, rapid, reversible and incremental actions and feedbacks [9]. Response to the input using touch inputs and virtual keyboards provides direct and smooth interfaces. Moreover, sensors like accelerometers are used for additional user actions such as changing device screen from landscape to portrait depending on the orientation of the device.

Android devices are opened on home screen that is the primary navigation and information point of the device. Home screens are generally made up of application icons and widgets. When the user touches an application icon, the interested application will open. On the other hand, widgets have the task of displaying live and updated content like weathercast or social media updates.

In Android Operating System, the status bar at top of the screen shows information about connection strength. Notification updates can be revealed by pulling down this status bar.

2.3. Android Applications

In Android Operating System, users can get application from Google Play, Amazon Appstore[3] or they can install the application's "APK" file from a third-party site. On devices following Google's compatibility requirements and having the license of Google Mobile Services software, Google Play application is pre-installed. In Play Store, more

[2] List of other recent Android Operating System features can be gathered from the link "http://en.wikipedia.org/wiki/List_of_features_in_Android".

[3] http://www.amazon.com/mobile-apps/b/ref=topnav_storetab_mas?ie=UTF8&node=2350149011

than one million applications available as of July 2013 and these applications have been installed 48 billion times as of May 2013 [10].

2.3.1. Platform

An Android application is a program that extends functionality of device. To create an Android application, Android Software Development Kit (SDK) and Java programming language is used. This SDK comes with debugger, emulators and documentations that composes development environment.

As it is mentioned in Section 2.1, Android builds on a Linux kernel and it has a layered environment as shown in Figure 1.

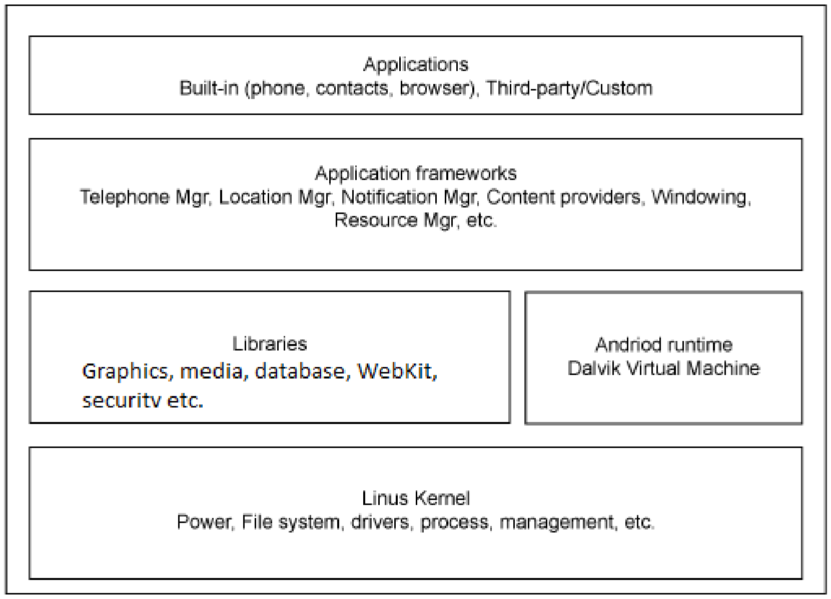

Applications
Built-in (phone, contacts, browser), Third-party/Custom
Application frameworks
Telephone Mgr, Location Mgr, Notification Mgr, Content providers, Windowing, Resource Mgr, etc.
Libraries
Graphics, media, database, WebKit, securitv etc.
Andriod runtime
Dalvik Virtual Machine
Linus Kernel
Power, File system, drivers, process, management, etc.

Figure 1 Android software layers (Adapted from [11])

Android applications that are coded in Java are run on a virtual machine called Dalvik Virtual Machine (DVM). This virtual machine is an open source technology and it differentiates from the Java Virtual Machines that are stack machines because of using a register-based architecture [9]. This architecture allows DVM to run on low memory and use its own byte code. As it shown in Figure 2, an Android application must run on DVM to turn into a Linux-kernel managed process.

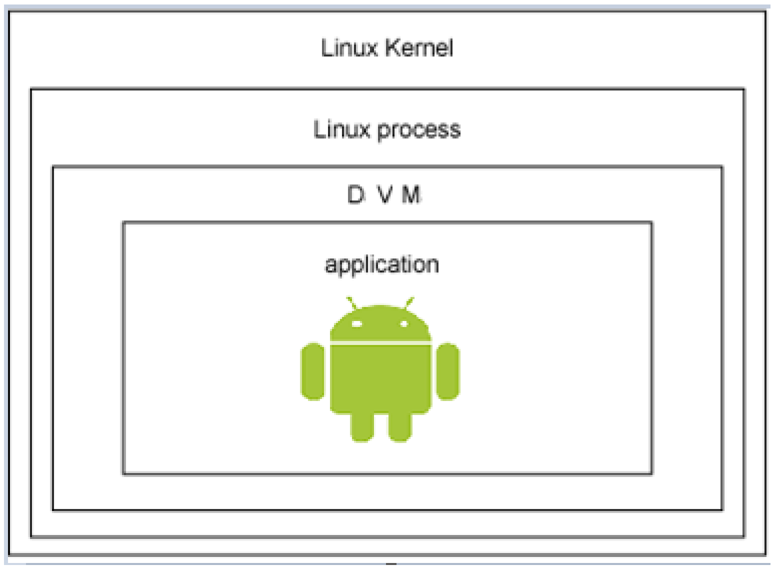


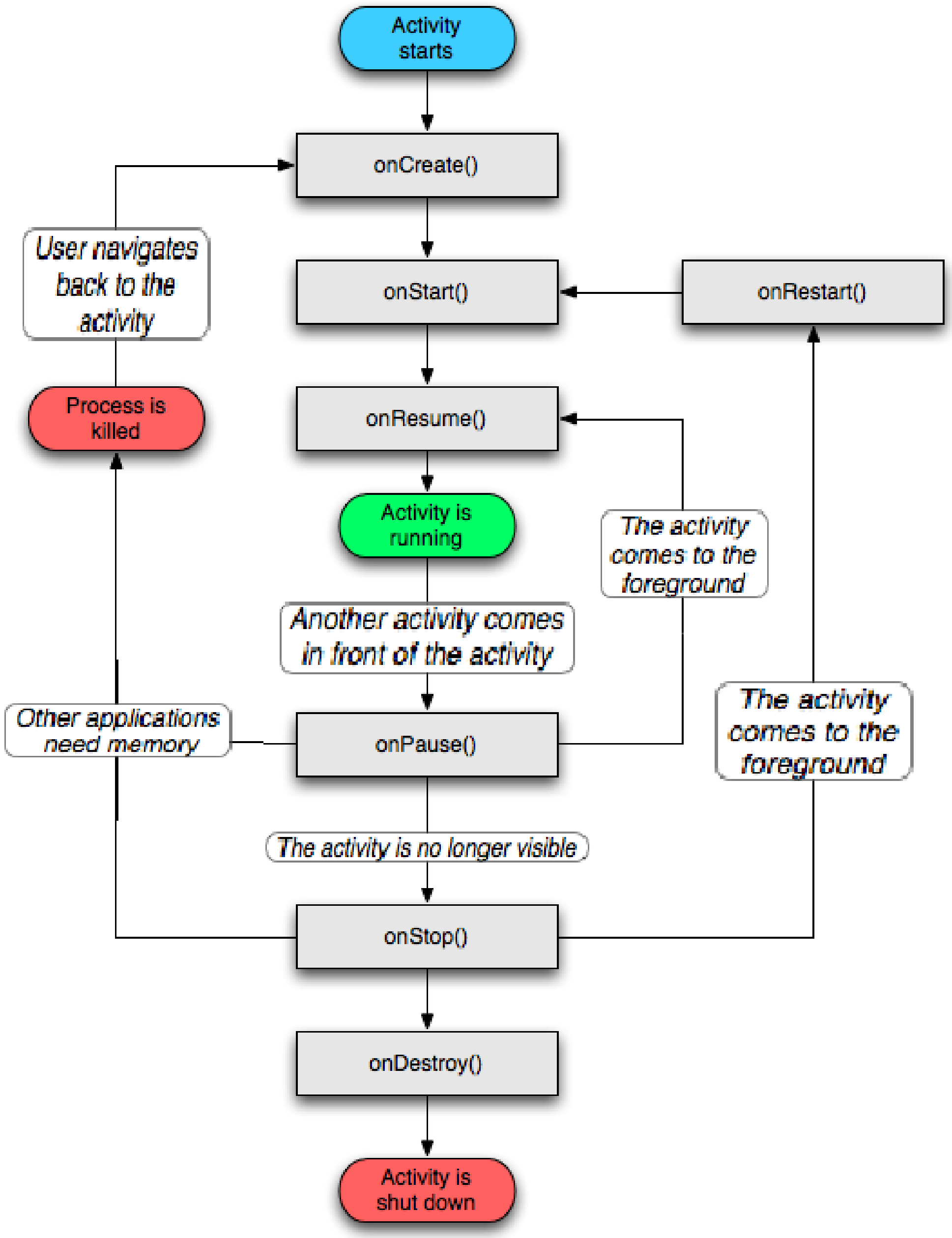
Figure 2 Stages that turns an Android application to Linux-Kernel managed process (Adapted from [11])

2.3.2. Application Development

When an Android application starts, the run time environment creates a “user interface thread”. In Android system, this thread is the “main thread”. This thread delivers the events to the proper user interfaces widgets and creates other threads.

An Android application consists of the following components:

- Activity: An activity is a class where the user interfaces of the applications are implemented. Activities can also be considered as tasks that a user can do in a screen. For example, sending a mail can be done in an activity and composing a mail can be done in another activity. Every application has a special activity named “main activity” which is the launcher of the application. [12]. An application contains multiple activities that are bound to each other usually by starting each other. When a new activity starts it is pushed to the stack which is called “back stack”. When user presses the back button, current activity popped from the “back stack” and previous activity is shown to the user. Normally, popped activities from the stack are destroyed and garbage collected. One of the situations the garbage collector cannot collect an activity instance, which is not a normal situation, is explained in Section 4.1.1. Managing an activity lifecycle is an important task for a developer. An activity instance in an application switches between different states, those states are “activity is foreground of the screen (active)”, “activity has lost focus but it is visible (paused)”, “activity is completely obscured by another activity (stopped)”, “when paused or stopped system drops the activity (destroyed)”. For each state, Android system calls a series of lifecycle methods [13]. As it is shown in Figure 3, “OnCreate” method is called when the activity is first started, “OnStart” method is called when the activity becomes visible to the user, “OnRestart” method is called when the stopped activity is not destroyed and being started again, “OnResume” method is called when the activity will start interacting with the user, "OnPause" method is called when the system starts the previous activity, "OnStop" method is called when the activity is no longer visible, "OnDestroy" method is the final call before your activity is destroyed by the Android system [12].



Activity starts
onCreate()
User navigates back to the activity
onStart()
onRestart()
Process is killed
onResume()
Activity is running
The activity comes to the foreground
Another activity comes in front of the activity
Other applications need memory
onPause()
The activity comes to the foreground
The activity is no longer visible
onStop()
onDestroy()
Activity is shut down

Figure 3 Activity lifecycle methods (Adapted from [12])

- Context: Context is responsible for holding information about the environment. These components hold the current state of the activity or application and the global information about the application. They allow access to the application specific resources, classes and application-level operations such as launching activities [14]. Every activity object is a context and they are named as "context-activity". Another kind of context is "context-application" which is independent from other activities lifecycles and lives as long as the application runs.

- Service: A service is an application component that persists for a long-time and stays in the background that means it does not have a user interface. One of the most powerful properties of the services is that they can run in the background even if the user switches another application [15].
- Intents Intent is the object that is used for communicating between components of an application. There are three important usages of intents; to start an activity, to start a service, to deliver a broadcast [16].
- Content Providers: Content providers can be considered as the data servers managing access to a structured set of data. They are the standardization between processes that communicates each other for data [17].
- Broadcast receivers: A broadcast receiver is used for launching an application to respond to events that are sent to whole device range such as receipt of a text message or battery low notification.

CHAPTER 3

RELATED WORK

In this section, available tools and studies related with our work are listed and explained. Tools and studies in this section can be divided into three categories: Android static analysis tools, Android dynamic analysis tool and bad practice detectors. The first one discusses the tools that analyze source code of the Android applications to find common mistakes without executing the application. The second one discusses the tools that test Android applications against the run-time problems like sudden crashes, slowness. The third one finds the bad practice patterns in Java source codes. Lastly, we give other bug finding studies that are not fit those three categories.

3.1. Android Static Analysis Tools

3.1.1. Android Lint

Android Lint is an Android project source code scanner tool for possible bugs and available as command line tool or *Eclipse*[4] and *IntelliJ*[5] plugin. Android Lint released in version 16 Android Development Tools (ADT) [18].

Type of errors checked by Android Lint

Android Lint checks the types of errors [18]:

- Missing and unused translations in localization files
- Performance problems in layouts
- Unused resources
- Problems related with internalization and accessibility like hardcoded strings
- Problems related to bitmaps like densities, wrong sizes
- Problems related to manifest

In addition, there is a list that explains all checks performed by Android Lint in ADT official site[6].

---

[4] http://developer.android.com/tools/debugging/improving-w-lint.html#eclipse

[5] http://blog.jetbrains.com/idea/2012/02/integration-with-android-lint-tool-in-intellij-idea-111/

[6] http://tools.android.com/tips/lint-checks

Architecture

In the structure of Android Lint, there are basically four components: issue registry, which is a container for issues, detectors, driver that drives other components and reporting system. This structure is shown in Figure 4 Structure of Android Lint

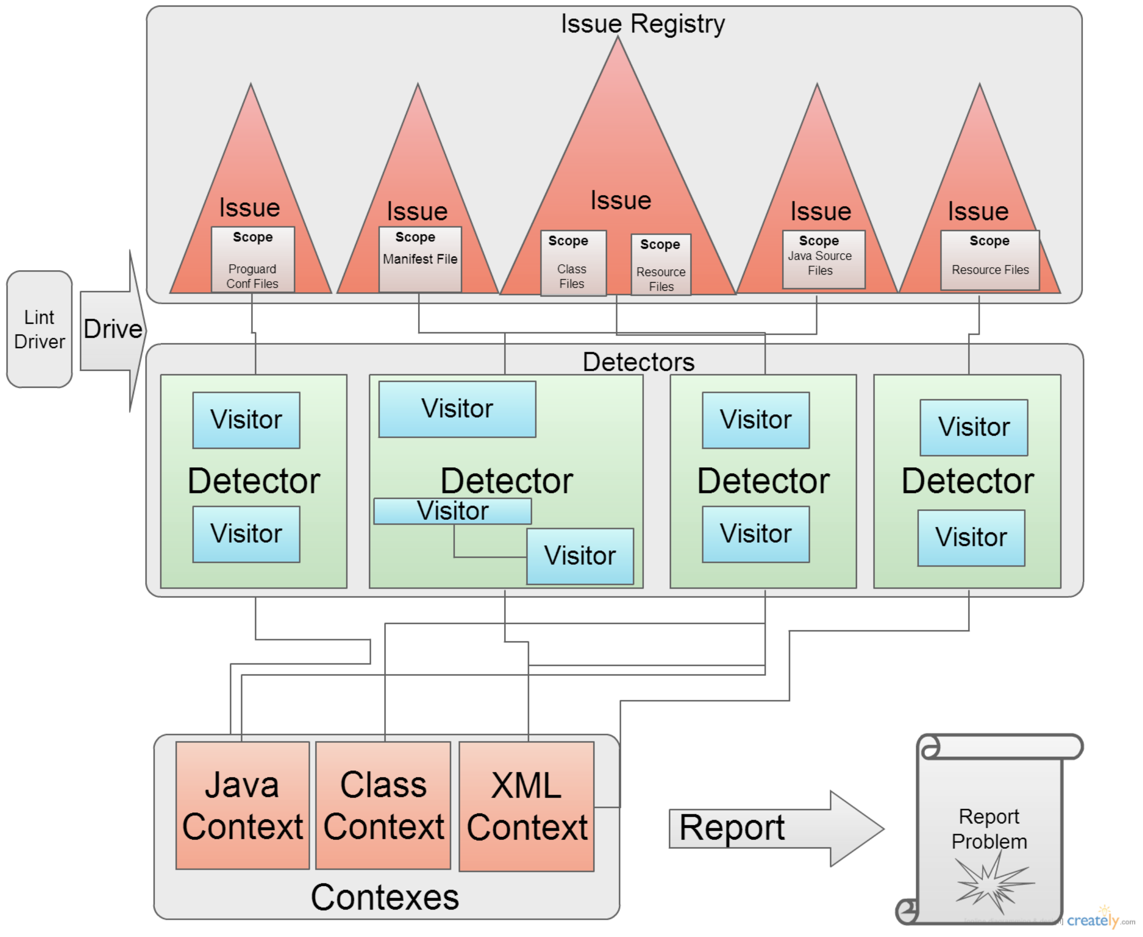


Figure 4 Structure of Android Lint

Android Lint allows its capabilities by extending its checks by using its API. `Detector`, `Scope`, `Issue` and `Visitor` are the important classes used for developing extensions. In addition, there is a reporting system to announce problems in the Android source code.

<u>Detectors</u>

A detector is a class that implements `Detector` interface to find a problem. When a detector finds a problem, it classifies this problem as an issue and reports this issue with information containing: explanation, how to solve the problem, priority and category. In addition to built-in detectors, one can implement custom detectors. A detector can implement one or more of these three interfaces: `detector.XmlScanner` to scan *.xml* files, `Detector.JavaScanner` to scan *.java* files, `Detector.ClassScanner` to scan *.class* files[7]. These interfaces indicate which file types the detector will examine.

Android Lint program processes detectors in a predefined order according to type of files the detector scans. The predefined processing order of detectors is:

1. Detectors scanning *Manifest File*
2. Detectors scanning *Resource files*,
3. Detectors scanning *Java sources*,
4. Detectors scanning *Java classes*,
5. Detectors scanning *Proguard files*.

The main class of the framework is called `LintDriver` class. This driver initiates each registered detectors to check the project against problems.

Issues & Scopes

An issue is a data class holding information about the problem found by a detector. Issues can be considered as the problems found by the detectors. An issue in Android Lint API is associated with a description, explanation, category and priority. The properties of the issues, such as priority of an issue, can be customized by the users by using preference files.

The relationship between detectors and issues is that a detector is responsible for scanning through the Android code, finding problems, creating issue instances for each of the problem and reporting them. The important point here is there can be more than one issue instance to report in a detector.

Each issue has a scope that specifies in which file type the problem may arise. A scope is an enumeration representation that lists parts of an Android project that are *Resource Files*, *Java Source Files*, *Class Files*, *Proguard Configuration Files* and *Manifest File*. For example, according to this API, if a detector finds a bug in pure java code file it must have an issue that is scoped with `Scope.JAVA_FILE_SCOPE` and must implement the `Detector.JavaScanner` to scan java files. [19]

Visitors

Detectors need to examine the source files and the framework provides visitors to parse and gather information on these files. The top-level visitor is an interface called `AstVisitor` defines the visit methods to traverse each node of the abstract syntax tree (AST). One concrete visitor provided by the framework is called `ForwardingAstVisitor`.

[7] Class diagram of those classes are shown in APPENDIX D Figure 29, Figure 30, Figure 31

A good way to implement a detector is extending the `ForwardingAstVisitor` and overriding the visit methods for the nodes of interest to find problems. The custom `Detector` will instantiate this extended visitor class when invoked by the LintDriver. The nodes of the AST are the visitables implementing the `accept` method shown in Figure 5.

```
1 public void accept(lombok.ast.AstVisitor visitor) {
2     if (visitor.VISIT_A_TYPE(this)) return;
3     for (lombok.ast.Node child : this.typeArguments.asIterable()) {
4         child.accept(visitor);
5     }
6     visitor.endVisit(this);
7}
```

Figure 5 Accept method

In the Figure 5, in `VISIT_A_TYPE` method, `TYPE` represents an AST node such as *if*, *VariableDeclaration*, *MethodDeclaration* or *Comment*. Here, the accept method implements the traversal algorithm: the visitor is ordered to traverse the children of the current node only if the condition in line 3 returns false. In `ForwardingAstVisitor` class, all of the visitor's `VISIT_A_TYPE` methods' calls return false and makes an automatic traversal in Android source code.

Reporting

When a detector finds a problem, it must call the `report` method on the context object. Context object is a subclass instance of a `Context` class in Android Lint API that can be `JavaContext`, `ClassContext` and `XmlContext` as shown in Figure 4 Structure of Android Lint. The `report` method is called with the parameters: *issue* instance that is found, *location* of the issue where the problem occurred, *message* to show the user and associated data.

3.1.2. PerfChecker and VeriDroid

In two of their studies, Liu at al works on bug patterns in Android developments. First, Liu at al created a tool that is named VeriDroid which detects null pointer de-references and resource leak defects. VeriDroid applied on 5 Android applications and it detected 7 bad-practice patterns" [20]. Second, they collected 70 performance bugs and create a tool that finds patterns that may cause of those performance bugs and named as PerfChecker. PerfChecker can find two patterns: "Potentially long running and not threaded operations like network, db etc. in main thread" and "Not reusing views in list views" [21]. After developing their tool, they applied the tool on 29 popular Android applications. As a result PerfChecker found 126 bad-practices and 68 of them admitted by the developers of the applications. Furthermore, 20 of those bad-patterns were easily resolved by means of the recommendations of the PerfChecker.

Those two works of Liu at al resemble our work in a way that they aim to find bad-practices; however, with Bad-Practice Finder we find different patterns that may cause performance bugs or other fallacies.

3.2. Android Dynamic Analysis Tools

3.2.1. UI/Application Exerciser Monkey

The Monkey (UI/Application Exerciser Monkey) is a program that executes on an emulator or a device and generates random user events such as clicks, touches, gestures or system-level events. This program can be used as a random, repeatable stress-tester for the Android applications while developing. [22]

When the Monkey executes, generated events are sent to system. The system under test is watched by The Monkey against three conditions. Three conditions and behavior of the Monkey are:

- Attempt to navigating other packages (if the Monkey is constrained running on one or more specific packages)
    - Behavior: the Monkey stops the attempt.
- Crashes and unhandled exceptions from application
    - Behavior: the Monkey stops and reports the error.
- ANR error from application
    - Behavior: The Monkey stops and reports the error.

Moreover, the Monkey has an option, which effects reporting verbosity on progress of the Monkey and generated events.

The Monkey tool only recognizes that there is problem in an Android application such as an Out of Memory exception. However, one cannot understand this exception caused by, for example, using non-static inner classes whose life is longer than its outer class, which is the situation explained in Section 4.1.1. As a result, the Monkey only gives the result not the reason behind the crush or slowness. On the contrary, our tool aims to find the reasons of *Out of Memory* exception, *ANRs* or *slowness* in the applications.

3.2.2. Systrace

Systrace analyzes performance of an Android application by capturing and displaying execution time of the application processes and other Android system processes. The data from Android kernel, such as CPU scheduler or disk activity are combined with the application threads by Systrace to create an HTML report. This report shows the overall picture of system processes of the Android device for a period of time. Systrace tool is useful in diagnosing problems related to display problems, which are slow drawings, stuttering in motion or animation.

Systrace tool makes a dynamic trace in the source code and gives a performance picture to the developer. Developers can see which threads of application uses CPU time more. However, this tool, like the Monkey, gives no information what may cause this overuse of CPU. Moreover, developer must do intuitive guesses like this thread must not use that much of CPU. As a result, Systrace can be used for demonstrating bug is existed and finding approximate location of bug in code. Of course, Systrace will prove that the threads that contain the bad-practices in 4.1 are problematic but will not provide any information about which one of them causes that problem.

3.3. Bad-Practice Detectors

3.3.1. FindBugs

FindBugs [23] is a bug pattern (bad-practice) detector for Java programs. Hovemeyer et al. gathered several bug patterns, both simple and complicated bugs, from real life programs. Surprisingly, those bugs appeared even in professional applications and libraries. They realized that there are a lot misused features of language or APIs of Java. In an experience with *Google* [24], using FindBugs, they filed more than 1700 bug reports and 640 of them fixed. This work made 1000 of the 4000 issues to resolve. Therefore, Hovemeyer et al. showed that even developers who specialized on Java might have significant gaps in their knowledge, which causes bugs. Finally, they concluded that tools for finding bugs help developers to find feature bugs by raising their awareness about subtle correctness issues. We think the same way and share this conclusion, therefore we developed a similar tool specialized for Android applications.

Both FindBugs and the tool developed in our study detect bug patterns and bad-practices. The context is, however, different. FindBugs is targeted toward general Java applications. On the other hand, our tool focuses on the bugs specific to Android applications.

FindBugs uses Java byte code to find bugs so program does not need source code of subjected Java program. List of bugs that this program can be detect are at [25]. These bugs can be classified in four categories [26]:

- Single-threaded correctness issue
- Thread/synchronization correctness issue
- Performance issue
- Security and vulnerability to malicious untrusted code

3.3.2. AMC: Verifying User Interface Properties for Vehicular Applications

This study [27] is another example that formalizes bad-practices, bug patterns and provides a tool that detects them automatically. The context of this bug detector is vehicular applications. The mobile applications (e.g. GPS route planner) in the vehicular environments are critical applications. The reason, that they are critical, is they must not distract driver from their primary task of operating the vehicle. There are studies providing best-practices that mobile application developers can follow. However, Lee et al. revealed that no application in Android marketplace followed those guidelines.

In the light of these facts, they developed a tool called AMC that explores the graphical user interface (GUI) of Android applications. AMC tool detects violations of vehicular design guidelines and gives developers early feedback. They applied their tool to 12 application and saw that the tool detect the existing or absent of 85% of guideline items with false positive and false negative rate of under 2%. For the remaining 15% cases, it helps reducing the number of application screens to check by an expert 95%.

3.4. Other Bug Finding Studies on Android

There are several bug finding studies in Android Applications and most of them uses static analysis. For example, [28] finds possible weaknesses containing security and privacy intrusions in communication between applications. In this work, Chin et al. realized that inter-application communication might have vulnerabilities and violates application security policies. To find these vulnerabilities, they provided a tool called ComDroid. ComDroid applied on 20 applications and it found 34 vulnerability that can be occurred while two applications are communicated. In addition, 12 of the applied 20 applications have at least one security vulnerability. ComDroid is specialized for one type of security bugs that includes intents. However, the fact that most of the applied 20 applications have at least one bug shows the importance of bug finders.

Another example, [29] detects a set of bugs and shows what is draining battery of mobile device. Pathak et al, creates a study that focus on smartphone energy bugs and they thought that this gives a way to new studies on that area. To do that, they categorized the energy bugs into three categories. Using dataflow analysis, they created detection solutions to energy bugs and implement these solutions in order to find energy bugs in run-time or compile-time. Similar methods such as categorizing the bugs and finding bugs using automated procedures is used in our study but we found bad-practices that causes that causes memory leaks or ANR dialogs which are out of scope for Pathak et al.

Last example, [30] reveals the unnecessary permissions included in manifest file of the applications. To achieve this goal, Felt et al created the tool named Stowaway that composes a set containing maximum permissions required for an application to run. After that, they used Stowaway for finding required permissions of 940 Android applications and matched this set against actually requested by manifest of each application. This experiment showed that over 300 of those applications have unnecessary privileges. Moreover, results indicates that developer confusion induced by API documentation errors and deficiency of knowledge causes those over-privileges. This work specializes to permission errors that may misleads the application users while installing an application. Even though they detect different bad-practices than our study, like the focus in this work, we also think that mistakes of developers may affect the user experiences.

All these studies are essential, useful but they have different focus then our study because they do not aim to find reasons for *Memory Leaks* or *ANR* errors

CHAPTER 4

METHODOLOGY

In this thesis, we collected bad-practices and their proposed solutions. For this step, we have gathered the experiences of Android developers and the common pitfalls they have faced over time that are published on the Internet and tutorials such as *Android Developers*[8].

After gathering this information, we have developed a bug-hunting tool that detects these bad-practices in a given Android application. The tool reports the problematic parts of the code and the effects of these problems such as *ANR* or *memory leak*. Next, we have collected a set of open source Android applications and run the bug-hunting tool on them. Finally, we have analyzed the result of the runs and examined the relationship between the user ratings (i.e. the reputation) of the applications with the number and type of bad-practices. In this chapter, we explain the first three steps in detail and the next chapter presents the analysis step.

4.1. Bad-Practices in Android development

To form a set of bad-practices that is recognized by the Android development community, we have examined tutorials, blog posts of experts and discussion forums such as *StackOverflow*[9], *Coderanch*[10] and *Android Developers*[11]. Furthermore, Google search is performed using the keywords *"Android programming best practices"*, *"Android programming bad-practices"*. Making this search reveals studies and presentations that are concerning *"User Interface Best and Bad-practices"*, *"Security Best and Bad-Practices"* and *"Performance Best and Bad-Practices"*. Practices that are intuitive and objective are eliminated and a list of bad practices whose detection can be automated with minimum user interaction are extracted. In total, we have examined over 31 web sites to extract the set of bad-practices. The list of the web sites is given in the APPENDIX A.

Since *Android Mobile Development* is quite popular in recent years, Internet is full of tutorials and blogs that teach about what to do and what not to do. Although there are

[8] http://developer.android.com/google/index.html

[9] http://stackoverflow.com/

[10] http://www.coderanch.com/forums

[11] http://developer.android.com/google/index.html

several code analyzers, for example *FindBugs* [23] and Android Lint [18], for bug detection and testing frameworks, there are still some bad-practices that are still need to be discovered manually by developers. The following list gives bad-practices that have maximum occurrence rate and can be found by using static analysis algorithms and they can cause serious problems and have solutions in these blogs and sites:

- Using non-static inner classes
- Not setting thread priorities
- Not using a cancellation policy in a thread
- Not reusing views in list view

The items in this list cause *Memory Leaks*, *Thread Leak* (a special kind of memory leak), *ANR* error (application not responding) and slowing the application.

The bad-practices we have gathered contain several more items and those extra items are given in APPENDIX B; however, for this study, it is important that they are easily detected and well defined. In addition, the items in this list cover most of the common causes that freezes or crashes the application and makes the Android system offer the user to terminate the application.

Next, we explain each of these four bad practices.

4.1.1. Using Non-Static Inner Classes

In Java language, an *inner class* is a class nested within another class. When an inner class object is created, that instance holds an implicit and hidden reference to the outer class` instance. This hidden reference is eligible for garbage collection purposes. Compilation makes these inner classes seem to be just like the ordinary classes; the hidden reference becomes real and the only connection between inner and outer class will be name of the inner class. [31]

Garbage collection is a mechanism to claim memory that not accessible by the running program (main thread or any other live threads). A *Garbage collector* periodically disposes of objects that are eligible for garbage collection using mark-and-sweep algorithm. Eligible, in here, means, *"There are no references to the object"*. In Java language, whenever an inner class instance is created, because of the nature of their definition, that inner class makes a reference to outer instance. This means that there is no inner class instance without an outer class. This situation effects the garbage collector when it needs to reclaim the memory for an instance of the outer class. The collector cannot reclaim the memory of the outer instance even if it is not referenced by any object other than its inner instance. This case occurs in Android development when inner classes of *Activity* is created, which is a common practice. When the lifetime of an inner class instance is longer than lifetime of the *activity* instance, due to the implicit reference, the activity will not get garbage collected causing a *memory leak*. This is an important issue because Android make use of a lot of anonymous inner classes like `Runnable`, `AsyncTask` and `Handler`.

To understand how the memory leak occurs, consider the following example in Figure 6.

```
public class Outer {

    private String someInstanceVariable = "";

    private InnerClass anonymous = new InnerClass() {

        public void someMethod() {
            someInstanceVariable = "Variable";
        }
    };
}
```

Figure 6 Outer class source code

When the code sample in Figure 6 is compiled, compiler results two class files: *Outer.class* and *Outer$1.class* and the *$1* in here means the implementation of anonymous inner class. If a disassembler used, the extended source code in Figure 7 will show the actual code that is created by the compiler.

```
1  class Outer$1 implements InnerClass
2  {
3
4     final Outer this$0;
5
6     Outer$1(Outer paramOuter) {
7        this$0 = Outer.this;
8     }
9
10    public void someMethod()
11    {
12      Outer.access$0(Outer.this, "Variable");
13    }
14  }
15
16 public class Outer
17 {
18  private String someInstanceVariable = "";
19  private InnerClass anonymous = new Outer.1(this);
20
21  static /* synthetic */ void access$0(Outer outer, String string) {
22        outer.someInstanceVariable = string;
23  }
24 }
```

Figure 7 Outer class decompiled code

Important thing here is inner class *Outer$1* here holds a reference to Outer instance (see line 7) by using a static *access$0* method. In this case, because of access$0 method and Outer.this variable, the garbage collector cannot claim the memory for an Outer instance even if it is not referenced by any other objects.

Figure 8 shows a more clear and related example from an Android case. In this example, activity instance is implicitly referenced two times from its inner classes. One of them is `anOnClickListener`, which is an OnClickListener implementation, and the second one is `ATaskLongRunning`, which is an `AsyncTask` implementation. This means there might be two critical sections that an event like "orientation change" can possibly leak activity instance.

```
public class ActivityWithTask extends Activity {
    private Button aButton;
    private View.OnClickListener anOnClickListener = new View.OnClickListener() {

        public void onClick(View view) {
            new ATaskLongRunning().execute();
        }
    };
    private class ATaskLongRunning extends AsyncTask<Void, Void, Boolean> {

        @Override
        protected Boolean doInBackground(Void... voids) {

            try {
                // Just sleep.
                Thread.sleep(30000);
            } catch (InterruptedException e) {
            }

            // finish your work.
            return true;
        }

        @Override
        protected void onPostExecute(Boolean aBoolean) {
            aButton.setText(aBoolean ? "Great" : "Not Good");
        }
    }
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        aButton = (Button) findViewById(R.id.btnSave);
        aButton.setOnClickListener(anOnClickListener);
    }
}
```

Figure 8 An acivity source code example containing async task

Consider an orientation change by rotating the device. In such case, `anOnClickListener` instance will be destroyed with the activity object as long as it does not contain a long running code. However, the case of the `ATaskLongRunning` instance is different. As long as this task continues to run in background, it keeps a reference to old activity object even if the run time environment created a new `Activity` instance. Here, the old activity instance cause memory leak. As a result, using codes resembling the one in Figure 8 is one of the main causes of *Out of Memory Exception*.

To prevent from this issue there are four rules [32]:

- Remove the long-lived referenced to a context-activity.
- Garbage collector must not be considered as an insurance against memory leaks.
- Context-application must be preferred over context-activity.
- Non-static inner classes must be avoided in an activity if the programmer do not know when their life end. Static inner classes with weak references must be used in the activities.

4.1.2. Not Setting Thread Priorities

When an application starts on a device or emulator that contains an Android Operating System, a thread called *UI thread*[12] (also known as the *main thread* in this context) is created [33]. This thread is very significant because it is responsible for sending the events to the proper widgets. Mostly, those widgets consist of drawing events that are visible to users. For example, when a user touches a button on the screen, this thread sends the touch event to the widget that makes the buttons state to *pressed* and sends an invalidate request to event queue. After all the process is done, this thread dequeues the events containing request and sends a signal to the widget to draw itself.

Making the Android application multithreaded is essential for the developers to improve performance. Performing long tasks like database and network tasks blocks the main thread and this cause display to freeze and no event can be dispatched while this long task is underway. This situation is reflected to the user as the application stalls. Moreover, if the application hangs more than 5 seconds user faces with a screen that says your application is not responding (ANR). [33]

```
class ARunnable implements Runnable {
    /*
     * Define the logic for the thread
     */
    @Override
    public void run() {
        // Put current thread in to background by reducing its priority.

android.os.Process.setThreadPriority(android.os.Process.THREAD_PRIORITY_BACKGROUND);

        // TODO : do the rest of the work
    }
}
```

Figure 9 A runnable example that sets its priority

Aside from those multithreading lessons, there is another point that a developer should know about the multithreaded applications; when there is another thread in an application it competes with the main thread for the resources. To avoid this developers should lower the thread priority (as working in the background: `android.os.Process.THREAD_PRIORITY_BACKGROUND`) for the threads other than the main thread by using the method `android.os.Process.setThreadPriority`. This assignment must be at the beginning of the run method like shown in Figure 9.

If the thread priority is not set to a lower priority like shown in Figure 9, the probability of creating a slow application is increased. The reason behind this is that, the task in runnable operates at the same priority as the main thread by default. This equality in priorities makes main thread stop dispatching the user interface events, which tends to lags in screens. Even, not setting thread priorities may cause ANR (application not

[12] http://android-developers.blogspot.com/2009/05/painless-threading.html

responding) error. For example, consider the case where the main thread is given many tasks that must use a certain resource of the device. At the same time, a thread that does network operation begins and never yields that certain resource for a long time. Because priorities of two threads are equal, when they need the same resource, network thread may keep this resource and the main thread may not continue its execution until the network thread finishes its job. This situation delays all of the tasks including dispatching of user interface events and user may not get answer from the application for a while.

4.1.3. Not Using a Cancellation Policy in a Thread

Garbage collection mechanism in Java has a concept named GC Roots which are the objects that are referenced by the Java Virtual Machine (corresponds to Dalvik Virtual Machine in Android) itself. In a simple Java application there four kinds of GC Roots: Local Variables, Static variables, JNI References and Active Java Threads. [34]

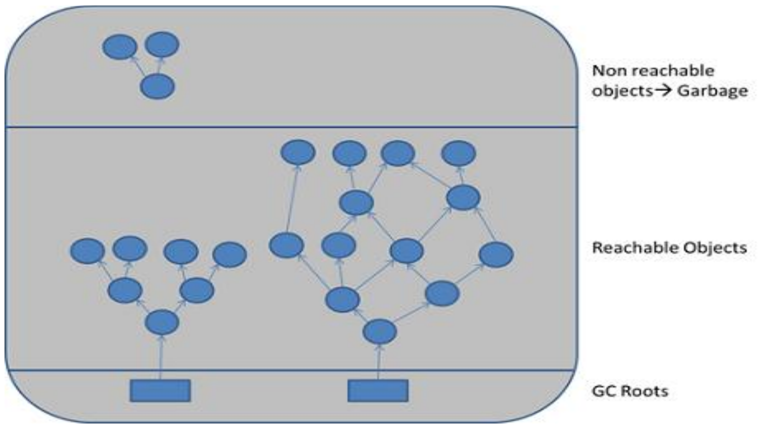


Figure 10 Java Garbage Collector roots (Adapted from [34])

Because threads are GC Roots, Dalvik Virtual Machine keeps hard references to all active threads in Android system. If a thread left running in a long time by setting up a `while(true)` or a similar mechanism, they may never be eligible for garbage collection. A developer must not assume that garbage collector collects the threads. For example, in the case shown in Figure 11 it is easy to think that after closing the application or the activity, that activity instance and any running thread associated with that activity are finalized and eligible for the garbage collection. However, generally this is not the situation. Java threads are living objects until all its process is done. Furthermore, this is the case until those threads are externally closed or the entire process is killed by the Android system.

As a result, it is important to implement a closing or cancellation policy for a thread and make use of this mechanism by regulating it with the activity life cycle. For example, in Figure 12 using a mechanism by calling `close` method in `onDestroy` method prevents accidental *thread leaks* from an application. [35]

```
public class ThreadLeakedActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        new AThread().start();
    }

    private static class AThread extends Thread {
        @Override
        public void run() {
            while (true) {
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

Figure 11 An activity example leaking thread

```
public class ThreadNotLeakedActivity extends Activity {
    private AThread aThread;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        aThread = new AThread();
        aThread.start();
    }

    private static class AThread extends Thread {
        private boolean running = true;

        @Override
        public void run() {
            while (running) {
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }

        public void close() {
            running = false;
        }
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
        aThread.close();
    }
}
```

Figure 12 An activity example not leaking thread

4.1.4. Not Reusing Views in List View

Showing lists is very important issue in Android development because it is a critical performance area for a resource constrained mobile device. The underlying reason for the problem is that there may be many items in a list. However, this must not prevent scrolling being smooth and fast and the process should not drain battery and not overuse the resources like CPU or RAM. [36]

```
public class WrongListAdapter extends BaseAdapter {
    private LayoutInflater mInflater;
    @Override
    public int getCount() {
        // code to get total number of items in this list.
    }

    @Override
    public Object getItem(int position) {
        //code to get an item from a list.
    }

    @Override
    public long getItemId(int position) {
        // code to get id of an item.
    }

    @Override
    public View getView(int position, View convertView, ViewGroup parent) {
        View item = mInflater.inflate(R.layout.activity_main, null);
        ((TextView) item.findViewById(R.id.txtLastSavedRecord)).setText(15);
        Bitmap mIcon1 = null;
        Bitmap mIcon2 = null;
        ((ImageView)
item.findViewById(R.id.action_settings)).setImageBitmap((position  &  1)  ==  1  ?
mIcon1 : mIcon2);
        return item;
    }
}
```

Figure 13 A list adapter example not reusing views

In Android Development API, there is a component called `ListView` that is designed for scalability and performance. It inflates its children, which can be named as *list item* and paints or prepares children that are or will become visible. While doing this operation developers must be careful about inflating the ListView with new views because creating new layout is very expensive operations.

Every time a `ListView` needs to show a list item on a screen, it calls `getView` method of its registered adapter. If the developers implement the adapter as in Figure 13 it creates a view for each invocation of `getView` method. This mechanism will fail because it drains the resources of the system.

```
public class CorrectListAdapter extends BaseAdapter {

    private LayoutInflater mInflater;

    @Override
    public int getCount() {
        // code to get total number of items in this list.
    }

    @Override
    public Object getItem(int position) {
        // code to get an item from a list.
    }

    @Override
    public long getItemId(int position) {
        // code to get id of an item.
    }

    @Override
    public View getView(int position, View convertView, ViewGroup parent) {
        if (convertView == null) {
            convertView = mInflater.inflate(R.layout.activity_main, parent, false);
        }
        ((TextView) convertView.findViewById(R.id.action_settings)).setText(15);
        Bitmap mIcon1 = null;
        Bitmap mIcon2 = null;
        ((ImageView) convertView.findViewById(R.id.btnSave)).setImageBitmap((position
& 1) == 1 ? mIcon1 : mIcon2);
        return convertView;


    `
```

Figure 14 A list adapter example reusing views

Instead of creating view each time, developer must reuse the views that are already created. Fortunately, Android has the ability to do that. As the example in Figure 13 shows, the `getView` method has three arguments `position` of item in list, a view `convertView` and a ViewGroup `parent`. The `convertView` is a recycled -if possible- view which was made invisible as the user pans the `ListView`. If a coder can use and update this view as it shows the new information instead of creating a new view, `ListView` only keeps views just enough to fill its space on screen and some additional recyclable views, even the adapter class instance has hundreds or thousands of items. A good example for doing that task can be seen in Figure 14.

In Figure 14 `getView` method contains a logic that checks `convertView` parameter against to be null and use it as a new view instead creating a new one.

4.2. The Detectors

In this thesis, to achieve the goal of hunting bad-practices in open source Android applications, a java program is developed called Bad-Practice Finder. This program takes an Android application source code as its input and searches for the pitfalls and bad-practices discussed in Section 4.2. The tool is created by using another open source program, Android Lint that is introduced in Section 3.1.1. Our tool can be considered as an extension of Android Lint, which aims to find the bad-practices, explained in Section 4.1 instead of its original bugs.

Android Lint is developed as a part of *Android Development Tools* and current version is *22.1.3*. To develop our tool Android Lint source code is downloaded in a form of three packages from the repository of Android Development Tools. These downloaded three packages are *lint-22.1.3-sources*[13], *lint-api-22.1.3-sources*[14], *lint-checks-22.1.3*-sources[15]. Downloaded source codes are compiled and architecture of Android Lint source code is analyzed for purpose of this study. Detailed information about Android Lint architecture is explained in Section 3.1.1.

To analyze an Android application, Android Lint API uses classes that implemented `Detector` interfaces. We implemented our custom detectors and we are able to scan the *classes, byte-codes* and *XML* files. To implement a detector first we must decide on its scope. The scope defines the types of files Android Lint will scan. Detailed information about scopes is given in Section 3.1.1.

For each entry in the Section 4.1, an issue and a detector class is created. These classes and their details are discussed in the following subsections.

When Android Lint checks an application, it looks for the issues in its registry. Currently there are 80 built-in issues, which are declared in the issue registry class named `BuiltinIssueRegistry` extending `IssueRegistry` class [37]. In order to focus only on the issues presented in this thesis, we removed the built-in issues from Android Lint by removing `BuiltinIssueRegistry` class. Instead of those registries, a new custom registry class `CustomIssueRegistry`, which register the issues that are owned by the detectors that hunts bad-practices explained in Section 4.1.

After changing the issues with the custom ones related with this project, the tool is executed on all open source project that we collected as discussed in Chapter 5.

4.2.1. InnerClassLeakDetector

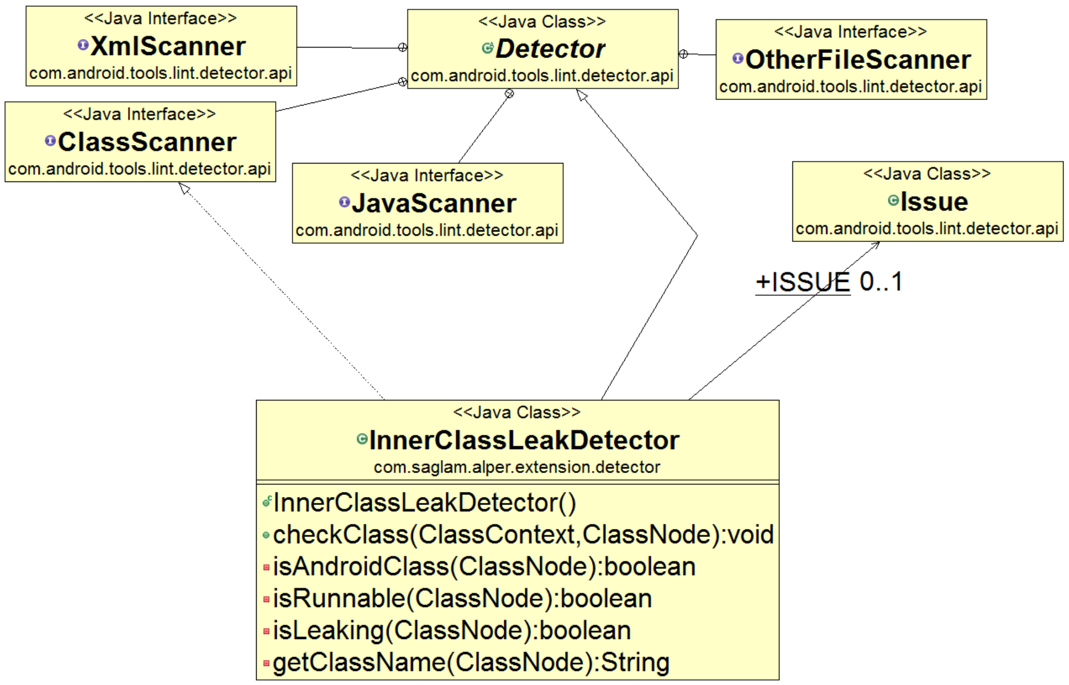
This detector finds the non-static inner classes that are explained in Section 4.1.1. This detector scopes (i.e. examines) all of the *.class* files. It extends `Detector` class and implements `ClassScanner` of Android Lint API (see Figure 15). This class overrides the `checkClass` method of the `ClassScanner` by applying the algorithm in Figure 16.

---

[13] http://search.maven.org/remotecontent?filepath=com/android/tools/lint/lint/22.1.3/lint-22.1.3-sources.jar

[14] http://search.maven.org/remotecontent?filepath=com/android/tools/lint/lint-api/22.1.3/lint-api-22.1.3-sources.jar

[15] http://search.maven.org/remotecontent?filepath=com/android/tools/lint/lint-checks/22.1.3/lint-checks-22.1.3-sources.jar



<<Java Interface>>
XmlScanner
com.android.tools.lint.detector.api
<<Java Class>>
Detector
com.android.tools.lint.detector.api
<<Java Interface>>
OtherFileScanner
com.android.tools.lint.detector.api
<<Java Interface>>
ClassScanner
com.android.tools.lint.detector.api
<<Java Interface>>
JavaScanner
com.android.tools.lint.detector.api
<<Java Class>>
Issue
com.android.tools.lint.detector.api
+ISSUE 0..1
<<Java Class>>
InnerClassLeakDetector
com.saglam.alper.extension.detector
InnerClassLeakDetector()
checkClass(ClassContext,ClassNode):void
isAndroidClass(ClassNode):boolean
isRunnable(ClassNode):boolean
isLeaking(ClassNode):boolean
getClassName(ClassNode):String

.

Figure 15 InnerClassLeakDetector class diagram

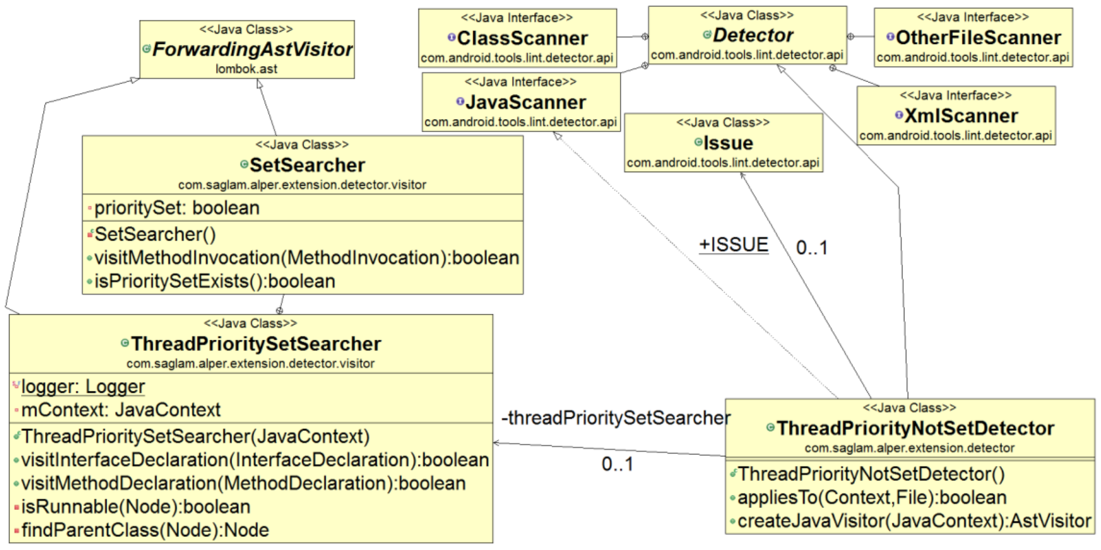
As it shown in Figure 16 Algorithm for finding leaking inner classes, the program first checks that if there is a “$” character in the name of the .class file. Second, it checks if this class is a built-in android classes like “Drawables” or “Filterables”. Third, because “Handler” class of Android is already checked by lint, it skips “Handler” classes. Lastly, if the class is not static and has a reference to its outer class program reports the issue with location.

```
function checkclass(classNode)
{
    if name of classNode does not contain "$" {
        return;
    }

    if classNode is an irrelevant android class {
       // Irrelevant Android classes are classes that do not cause
       // inner class leak and are built-in classes by Android Operating System
       // such as Drawables and Filterables
        return;
    }

    if classNode is android handler class {
       // Android handler class is already handled by Android Lint.
        return;
    }

    if classNode is not a static class{

        if classNode has a reference to outer class {

            report issue location in code with className
```

Figure 16 Algorithm for finding leaking inner classes

4.2.2. ThreadPriorityNotSetDetector

This detector finds the threads that do not set priority in its *run* method. This detector has the scope of *Java files*, which is achieved by using scope `Scope.JAVA_FILE_SCOPE` and implements the `JavaScanner` interface. This class creates and passes the java context to a visitor extending `ForwardingASTVisitor` which visits and searches for *runnables* and their `run` methods. This visitor invokes another visitor, `ThreadPrioritySetSearcher`, to search for the statement containing `setThreadPriority` method invocation in the body of run method. The class diagram of this detector is shown in Figure 17.



<<Java Class>>
ForwardingAstVisitor
lombok.ast
<<Java Interface>>
ClassScanner
com.android.tools.lint.detector.api
<<Java Class>>
Detector
com.android.tools.lint.detector.api
<<Java Interface>>
OtherFileScanner
com.android.tools.lint.detector.api
<<Java Interface>>
JavaScanner
com.android.tools.lint.detector.api
<<Java Interface>>
XmlScanner
com.android.tools.lint.detector.api
<<Java Class>>
Issue
com.android.tools.lint.detector.api
<<Java Class>>
SetSearcher
com.saglam.alper.extension.detector.visitor
prioritySet: boolean
SetSearcher()
visitMethodInvocation(MethodInvocation):boolean
isPrioritySetExists():boolean
+ISSUE
0..1
<<Java Class>>
ThreadPrioritySetSearcher
com.saglam.alper.extension.detector.visitor
logger: Logger
mContext: JavaContext
ThreadPrioritySetSearcher(JavaContext)
visitInterfaceDeclaration(InterfaceDeclaration):boolean
visitMethodDeclaration(MethodDeclaration):boolean
isRunnable(Node):boolean
findParentClass(Node):Node
-threadPrioritySetSearcher
0..1
<<Java Class>>
ThreadPriorityNotSetDetector
com.saglam.alper.extension.detector
ThreadPriorityNotSetDetector()
appliesTo(Context,File):boolean
createJavaVisitor(JavaContext):AstVisitor

Figure 17 ThreadPriorityNotSetDetector class diagram

4.2.3. ThreadNoCancelationPolicyDetector

This detector finds the bad-practice pattern mentioned in Section 4.1.3 which is not using a cancellation policy in a thread. It creates a visitor which searches for a cancellation policy in the runnable classes that have possibility to run forever (especially while loops) and checks that if this cancellation policy exists.

While creating an algorithm to detect cancellation policy absense in threads, we know that we could not cover all the cases. Because finding if there is a method which definitely closes a thread (i.e. developing an algorithm that checks if a thread will terminate) is an undecidable problem [38]. Therefore, we needed a heuristic approach to detect at least a subset of bad-practice occurences. Our heuristic approach involves the obvious ones which uses while with static conditions like “true” or “!false” and there is no exit statement in its block or there is an exit statement in its block but the condition which triggers this exit statement is not changed in any other method. This approach confirms that if there is a sharp “Not using a cancellation policy in a thread” bad-practice, our algorithm finds it. However, if the algorithm cannot find it, this does not mean that there is no “Not using a cancellation policy in a thread” bad-practice in the code. The psuedocode for this approach is shown in Figure 18.

```
global staticCondition = false;
global isThereBreak = false;
global closeMethod = false;
global breakCondition;


function visitWhile(whileNode)
{
    whileCondition = get condition from whileNode

    if whileCondition is "true" or whileCondition is "!false"
        staticCondition = true; // this means while(true) or while(!false)
                                // which can execute forever

    foreach children of whileNode
        if children is "break" or "return"
        {
            isThereBreak = true; // means while can be exited
            breakCondition = get condition from parent of "break" or "return";
        }
}

function visitMethods(methodNode)
{
    if(staticCondition and isThereBreak and (breakCondition is boolean))
    {
        foreach children in methodNode
            if children assignes breakCondition to false
            {
               closeMethod = true;
               return;
            }

    }
}
```

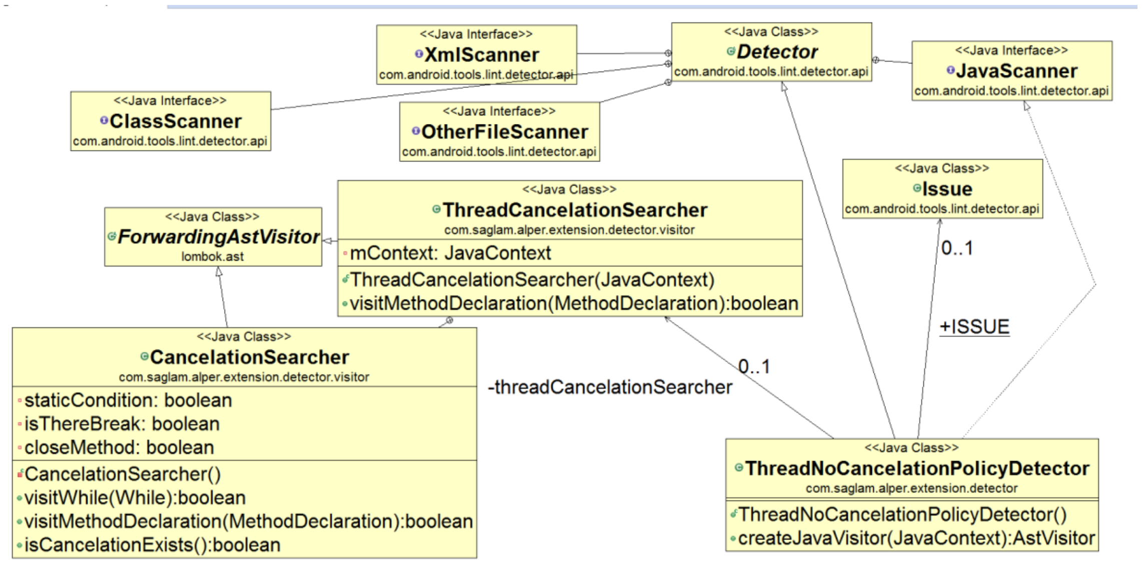
Figure 18 Algorithm applied in “visitWhile” and “visitMethods” methods of the visitor in ThreadNoCancelationPolicyDetector

In the algorithm shown in Figure 18, first while conditions are checked if they are static conditions like “true” or “false”. Second, among the statements in the while block, “break” or “return” statements are searched for whether there is an exit point from the while block. After finding static conditions and break conditions, the methods of the current class are visited to find the method that triggers the break statement via visitMethods function. After the `while` statements and runnable methods are visited, the algorithm shown in Figure 19 runs to check whether there is a cancellation policy or not.

```
function doesCancellationExist()
{
    if (closeMethod) {
        // only true when !staticCondition and isThereBreak is true
        return true;
    } else if (isThereBreak && (breakCondition is not boolean)) {
        // means while depends on a variable which is not a boolean,
        // no need to check if it runs forever.
        return true;
    } else if (staticCondition && !closeMethod) {
        // means there is a probability to run forever, because there is a break
        //condition which is Boolean, and this condition is not changed in any
       //other method
        return false;
    } else if (staticCondition && !isThereBreak) {
        // there is no probability to escape from the while loop.
        return false;
    } else {
        return true;
    }
}
```

Figure 19 Algorithm for deciding whether the cancellation policy exsists or not

This detector also has Java file scope and implements `JavaScanner` interface and has two visitors "ThreadCancellationSearcher" and "CancellationSearcher" which implements the algorithm presented in Figure 18 and Figure 19.



<<Java Interface>>
XmlScanner
com.android.tools.lint.detector.api
<<Java Class>>
Detector
com.android.tools.lint.detector.api
<<Java Interface>>
JavaScanner
com.android.tools.lint.detector.api
<<Java Interface>>
ClassScanner
com.android.tools.lint.detector.api
<<Java Interface>>
OtherFileScanner
com.android.tools.lint.detector.api
<<Java Class>>
Issue
com.android.tools.lint.detector.api
<<Java Class>>
ForwardingAstVisitor
lombok.ast
<<Java Class>>
ThreadCancelationSearcher
com.saglam.alper.extension.detector.visitor
mContext: JavaContext
ThreadCancelationSearcher(JavaContext)
visitMethodDeclaration(MethodDeclaration):boolean
0..1
+ISSUE
<<Java Class>>
CancelationSearcher
com.saglam.alper.extension.detector.visitor
staticCondition: boolean
isThereBreak: boolean
closeMethod: boolean
CancelationSearcher()
visitWhile(While):boolean
visitMethodDeclaration(MethodDeclaration):boolean
isCancelationExists():boolean
-threadCancelationSearcher
0..1
<<Java Class>>
ThreadNoCancelationPolicyDetector
com.saglam.alper.extension.detector
ThreadNoCancelationPolicyDetector()
createJavaVisitor(JavaContext):AstVisitor

Figure 20 ThreadNoCancellationPolicyDetector class diagram

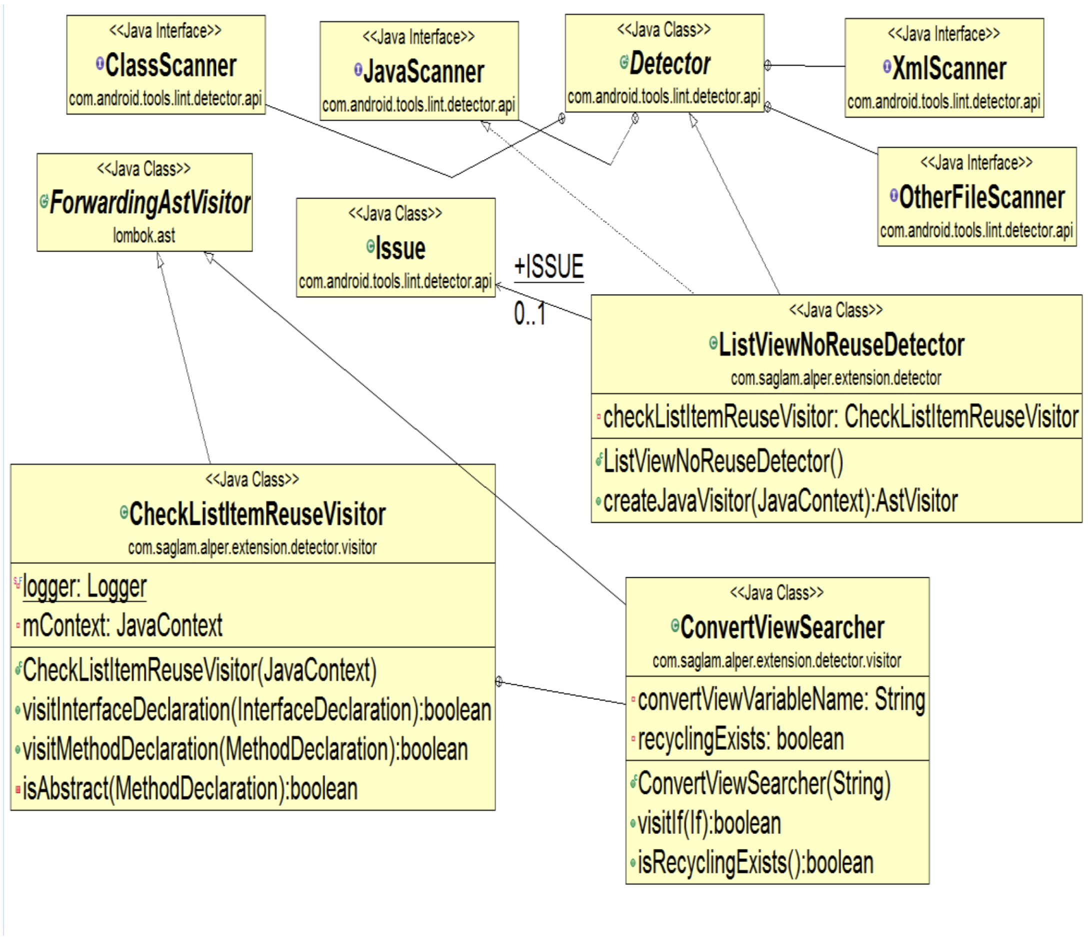
4.2.4. ListViewNoReuseDetector

Similar to the ThreadPriorityNotSetDetector and ThreadNoCancelationPolicyDetector this detector has a scope of Java files. This detector initiates a visitor that visits `getView` methods of list adapters and search for view-reusing in them. This detector applies the algorithm in Figure 21. The algorithm, first, looks for list view adapter classes and looks for the `getView` methods in these classes. In these methods, if the parameter, which may be reused, is checked against null value and a layout inflation occurs in this condition, the algorithm says that there is a conditional inflation. In the meantime, if the same parameter is checked for non-null and this parameter is modified in that condition the algorithm says that there is a reusing view. If at least one of these two conditions (conditional inflation and reusing view) is false in a `getView` method, then there is no recycling in that list view.

```
global isRecyclingExist = false;

function visitMethod(methodNode)
{

    conditionalInflation = false;

    classOfMethod = get class of methodNode
    if(classOfMethod is a listView Adapter)
    {
        methodName = get methodName from methodNode

        if methodName is "getView"
            convertView = get convertView parameter from methodNode
            foreach child in childiren in methodNode
            {
                if(convertView is check for null)
                    if inflation occurs
                        conditionalInflation = true;

                if( convertView is checked for non-null)
                    if convertView is modified
                        if convertView returned from getViewMethod
                            reusingConvertView = true;
            }

        isRecyclingExists = conditionalInflation && reusingConvertView
    }
}
```

Figure 21 Algorithm applied in "visitMethod" method of visitor in ListViewNoReuseDetector

In Figure 22, the class diagram of this detector is shown. The detector has two different visitors that are extended from "ForwardingAstVisitor". First visitor, "CheckListItemReuseVisitor" class traverses the classes looking for `getView` declarations. Second visitor, traverses body of the `getView` methods by applying algorithm explained in Figure 21.



<<Java Interface>>
ClassScanner
com.android.tools.lint.detector.api
<<Java Interface>>
JavaScanner
com.android.tools.lint.detector.api
<<Java Class>>
Detector
com.android.tools.lint.detector.api
<<Java Interface>>
XmlScanner
com.android.tools.lint.detector.api
<<Java Interface>>
OtherFileScanner
com.android.tools.lint.detector.api
<<Java Class>>
ForwardingAstVisitor
lombok.ast
<<Java Class>>
Issue
com.android.tools.lint.detector.api
+ISSUE
0..1
<<Java Class>>
ListViewNoReuseDetector
com.saglam.alper.extension.detector
checkListItemReuseVisitor: CheckListItemReuseVisitor
ListViewNoReuseDetector()
createJavaVisitor(JavaContext):AstVisitor
<<Java Class>>
CheckListItemReuseVisitor
com.saglam.alper.extension.detector.visitor
logger: Logger
mContext: JavaContext
CheckListItemReuseVisitor(JavaContext)
visitInterfaceDeclaration(InterfaceDeclaration):boolean
visitMethodDeclaration(MethodDeclaration):boolean
isAbstract(MethodDeclaration):boolean
<<Java Class>>
ConvertViewSearcher
com.saglam.alper.extension.detector.visitor
convertViewVariableName: String
recyclingExists: boolean
ConvertViewSearcher(String)
visitIf(If):boolean
isRecyclingExists():boolean

Figure 22 ListViewNoReuseDetector class diagram

4.3. Evaluating the tool Bad-Practice Finder

After developing all detectors, Bad-Practice Finder tested to confirm that if it satisfies all the expected requirements. To achieve that goal, a unit test consists of five applications that is planned to use in the experiments is applied to Bad-Practice Finder. Table 1 shows these five applications (CycleSteets, Gmote, Jamendo,Sasabus, Osciprimeics), their sizes in line of code, the number of bad practice for existed in each application, and the number of bad practices detected by our tool.

Table 1 Application's expected and detected bad-practice values

Application name	CycleStreets		Gmote		Jamendo		Sasabus		Osciprimeics	
Line Of Code	13686		27257		8709		8908		7036	
	Expected	Detected	Expected	Detected	Expected	Detected	Expected	Detected	Expected	Detected
"Not reusing views in list view" Count	6	6	1	1	1	1	0	3	0	0
"Using non-static inner classes" count	12	12	15	15	32	32	2	2	16	16
"Not using a cancellation policy in a thread" count	0	0	6	6	0	0	0	0	3	3
"Not setting thread priorities" count	0	0	7	7	4	4	0	0	18	18

Table 1 shows that there are no false positive or false negative result except from Sasabus application. In Sasabus application three false positive "Not reusing views in list view" bad-practice found. The reason is that, in some of their adapters, they used a different algorithm for reusing items in list view and this algorithm may not reuse the items (views) in some conditions that they previously define.

4.4. Program Performance

To show the performance of our Bad-Practice Finder tool, we applied it to the applications tested in Section 4.3. During this process, we measured total time and used maximum heap size using VisualVM[16] to execute Bad-Practice Finder on an application. The results of this experiment are presented in Table 2. In this table, the first column in shows the application name, the second column shows the size of the application, third column shows the maximum heap size used by our tool to examine the application, and the last column shows the time spent for the examination of the application.

Table 2 Bad-practice Finder tool performance metrics

Application Name	Line of Code (LOC)	Used Max Heap Size(bytes)	Execution Time(second)
CycleStreets	13686	223.465.072	8,695
Gmote	27257	56.028.520	5,075
Jamendo	8709	137.256.880	6,829
Sasabus	8908	154.105.288	6,671
Osciprimeics	7036	88.828.600	6,948

[16] *https://visualvm.java.net/?Java_VisualVM*

Among the applications in Table 2, the size of “Gmote” is bigger than the others in terms of LOC, but the memory (Used Max Heap Size) and time spent for this application is smaller than that of the other applications. This shows that there is no relation between the application size in terms of LOC and required memory and time to examine the application

CHAPTER 5

ANALYSIS

In the analysis part, we gathered statistical data of 100 open source Android applications. After collecting that data our “Bad-Practice Finder” tool is run on these applications to gather number of bad practices identified in Section 4.1. After getting this data, to show the relationship between the user ratings and the occurrence of the practices we used three different statistical approaches. This chapter presents our analysis and results.

5.1. Gathering Experiment Data

To gather open source applications, an application called *F-Droid*[17] that is a catalogue of Free and Open Source Software (FOSS) applications for Android platform is used. From F-Droid’s list of open source applications, which is listed in [39], we have collected average user rating for each of the application.

From F-Droid’s list of open source applications, which is listed in [39], applications are downloaded with their source code. To do that, all of the application unique package identifiers are saved to a file. A java program which uses *Selenium* [40] API is developed to download source code URLs. Algorithm of this program is shown in Figure 23. The program’s task begins with going each application’s page in F-Droid’s website that can be queried with application identifier. After that, the program gets HTML code of the self-page of application in F-Droid, parses the HTML code and selects the URL address of compressed source code. Lastly, the program downloads to source code to disk using URL address.

[17] https://f-droid.org/

```
baseaddress = "https://f-droid.org/repository/browse/?fdid=";

idList = read application ids from file;

for each id in idList
{

    appAddress= concatenate baseaddress and id;

    page = go to appAdress webpage;

    anchors = get all anchor elements links from page;

    foreach anchor in anchors
    {

        if( anchor text contains "source tarball")
        {
            tarBallLink = get href attribute of anchor;

            download tarBall from tarBallLink;

            break;
        }
    }
}
```

Figure 23 Algorithm for downloading application source code

In the set of F-Droid catalogue, there are approximately 1200 applications. We achieved to download the source code of 932 of those applications in the catalogue because some of the applications in the catalogue had some dead links or bad connections to their source codes. After downloading source codes, Google Play statistics are gathered from their official *Google Play* page for each application. These statistics are *count of for each rate option (1 to 5), average rate, total rate count* and *install count*. To achieve that goal, another program also using *Selenium* is developed. Algorithm of this program is shown in Figure 24. This program goes the each application's self-page in *Google Play* by querying with application identifier, gets HTML of page, parses HTML, extracts rating information using *tags-attributes* and *xpath* and saves them in a tab separated format. This operation eliminated 206 of those applications since these applications do not have Google Play self-page and we could not gather their user rating statistics.

```
baseaddress = "https://play.google.com/store/apps/details?id=";
idList = read application ids from file;

for each id in idList
{
    appAddress= concatenate baseaddress and id;

    page = go to appAdress webpage;

    revieverCount = get text of element with class-name "stars-count";

    point5Count = get text of element with class-name "bar-number-5";
    point4Count = get text of element with class-name "bar-number-4";
    point3Count = get text of element with class-name "bar-number-3";
    point2Count = get text of element with class-name "bar-number-2";
    point1Count = get text of element with class-name "bar-number-1";

    score = get text of element with class-name "score";

    numOfInstallElement    =    get    element    by    x-path    of    "id('body-
content')/div[4]/div/div[2]/div[3]/div[2]";

    numOfInstall = get "itemprop" attribute of numOfInstallElement;

    save data to as a line reviewerCount    +   '\t' +  point5Count + '\t' +
                        point4Count     +   '\t' +  point3Count + '\t' +
                        point2Count     +   '\t' +  point1Count + '\t' +
                        numOfInstall;
}
```
Figure 24 Algorithm for gathering application data form Google Play

368 of those applications in this catalogue cannot compiled because of the platform dependencies, missing jar files, missing manifest files, missing dependent projects, hard coded or absolute paths to personal computer locations. Rest of the applications in the catalogue are filtered with the conditions that

- Total rate count must be over 10
- Install count must not be under 500
- Line of code must not be under 500

After filtering them, to have the same distribution with the whole data, 100 of them with the same average user-rating distribution is chosen. APPENDIX C shows identifications of downloaded applications and their statistical data

Table 3 shows the average size of the subject applications in terms of line of code, minimum and maximum size of the applications. Recall that we also collect rate count for each application. Table 3 also shows the average rate counts. For example, in our pool, an application is rated 5 points on the average 2046 times.

Table 3 Statistical information about subject applications

Statistic Type	Value
Average Line of Code	17720
Minimum – Maximum Line of Code	~0,5 k - ~31,5 k
5 Point Average	2046
4 Point Average	558
3 Point Average	223
2 Point Average	106
1 Point Average	223

5.2. Analysis

In this section, the output of the Bad Practice Finder on the open source applications are analyzed and discussed. To investigate the relationship between the user ratings and bad-practice occurrences we used three different approaches that are explained in the following subsections.

5.2.1. Correlation between User Ratings and Bad-Practice Count

In this analysis, we first applied a min-max normalization on all of the user rating types (5 point, 4 point, 3 point, 2 point, and 1 point) and two composite user rating types: higher rates (sum of point 5 rate and point 4 rate) and lower rates (sum of point 2 rate and point 1 rate).

After normalizing the user ratings, we calculated *Pearson product-moment correlation coefficient*[18] between bad-practice counts and these normalized user ratings. The output extracted from this calculation is shown in Table 4.

In Table 4, the columns under “Rate Count” header show the value of *Pearson product-moment correlation coefficient value* between normalized rate counts of 5, 4, 3, 2 and 1 points. “Higher Rate Count” column and “Lower Rate Count” show the same coefficient values correspondingly for higher rates and lower rates. The coefficient values followed by an asterisk are significant because the p-value, shown below of the coefficient value, is smaller than 0.05.

[18] http://en.wikipedia.org/wiki/Pearson_product-moment_correlation_coefficient

Table 4 Pearson product-moment correlation coefficient between normalized rate counts (5 point, 4 point, 3 point, 2 point, 1 point, higher rates, lower rates, install count) and bad-practice counts ('' means significant correlation value which has a p-value of smaller than 0.05)*

Bad Practice Type	Rate Count					Higher Rate Count	Lower Rate Count
	5	4	3	2	1		
Not using a cancellation policy in a thread	0.251*	0.309*	0.364*	0.372*	0.444*	0.265*	0.428*
	p:0	p:0.012	p:0.002	p:0	p:0	p:0.008	p:0
Using non-static inner classes	0.284*	0.323*	0.285*	0.263*	0.169	0.295*	0.199*
	p:0.004	p:0.001	p:0.004	p:0.008	p:0.092	p:0.003	p:0.047
Not reusing views in list view	0.225*	0.250*	0.221*	0.216*	0.140	0.233*	0.164
	p:0.024	p:0.012	p:0.027	p:0.031	p:0.165	p:0.02	p:0.102
Not setting thread priorities	0.392*	0.414*	0.427*	0.478*	0.534*	0.404*	0.532*
	p:0	p:0	p:0	p:0	p:0	p:0	p:0

Table 4 shows that p-values of the correlations between "Not using a cancellation policy in a thread" - "Using non-static inner classes" bad-practices and the rate counts are smaller than 0.05 and the corresponding correlation coefficient values are significant. In addition, it is seen that those two bad-practices has more positive relationship with unfavorable points (which are closer to point 1) than the favorable points (which are closer to point 5). This fact means if "Not using a cancellation policy in a thread" - "Using non-static inner classes" bad-practices exist; users tend to give unfavorable points. This inclination also can be seen from the case that their significant "Higher Rate Counts" are smaller than their "Lower Rate Counts".

The other two bad-practices have insignificant correlation coefficient values as seen in the table. Therefore, we cannot comment about their relationship with the user ratings counts.

5.2.2. Relation with Categorization

For this analysis, we demonstrate the bad-practice counts and user ratings with box-plots. Box plot is used to demonstrate groups of numerical data and their variability with quartiles [41]. To create a box-plot representation of the experimental data results, we categorize the applications based on their weighted average rates, which show success of the application. Table 5 shows the categorization of the applications. The table also shows number of applications for each of the category we have in our pool.

Table 5 Application success categories

Category	Condition	Application Count
Very Unsuccessful Applications	Average Rate < 3.5	9
Unsuccessful Applications	3.5 <= Average Rate < 3.8	13
Bad Applications	3.8 <= Average Rate < 4.1	20
Not Bad Applications	4.1 <= Average Rate < 4.4	21
Successful Applications	4.4 <= Average Rate < 4.8	28
Very Successful Applications	4.8 <= Average Rate	9

After categorization of the applications, we created box-plot graphics of each bad-practice count versus categories using *SPSS*[19]. These graphics are shown in Figure 25, Figure 26, Figure 27 and Figure 28. To examine the significance of those graphics and categorization we apply the *Kruskal–Wallis*[20] test to the data. Result of this test is shown in Table 6 *Kruskal-Wallis* test on bad-practices count and Success category. The results are discussed below.

Table 6 Kruskal-Wallis test on bad-practices count and Success category

Graphic Name	Significance Value
"Using non-static inner classes" count vs Success category	0.018
"Not setting thread priorities" count vs Success category	0.069
"Not using a cancellation policy in a thread" count vs Success category	0.358
"Not reusing views in list view" count vs Success category	0.003

[19] http://www-01.ibm.com/software/analytics/spss/

[20] http://en.wikipedia.org/wiki/Kruskal%E2%80%93Wallis_one-way_analysis_of_variance

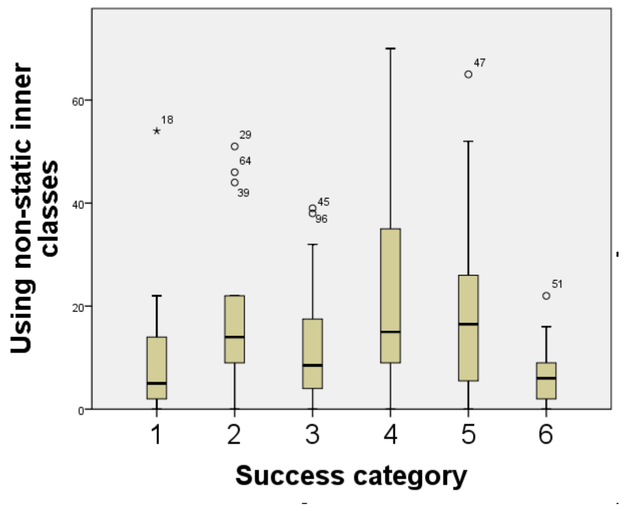


Figure 25 Box-Plot of "Using non-static inner classes" count vs Success category

Figure 25 shows that "Using non-static inner classes" fluctuates until the 4th category (Not Bad Applications) and there are a lot of outlier values in 1st, 2nd and 3rd success categories. However, in the "Not Bad Applications", "Successful Applications" and "Very Successful Applications" quartiles of the boxes are declined. This means that there is a reverse relationship between the successfulness of the application and the "Using non-static inner classes". Applying Kruskal-Wallis test with "Using non-static inner classes" and Success category gives a significance value, which is smaller than 0.05, that can be seen from Table 6 Kruskal-Wallis test on bad-practices count and Success category. This means that this box-plot is significant.

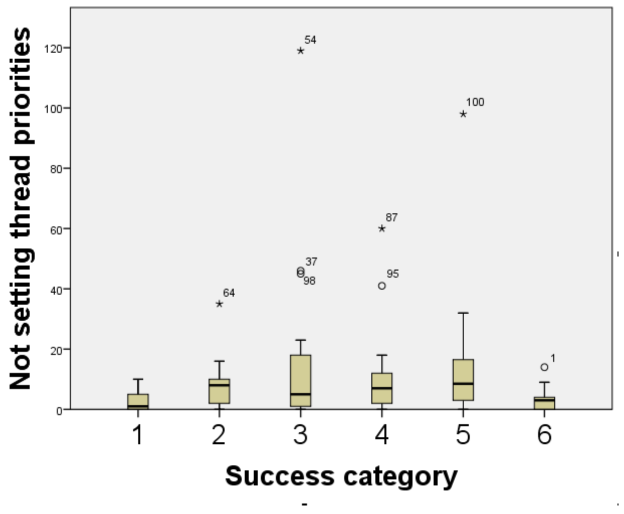


Figure 26 Box-Plot of "Not setting thread priorities" count vs Success category

The box-plot in Figure 26 Box-Plot of "Not setting thread priorities" count vs Success categoryshows the relation between "Not setting thread priorities" and Success category.

The boxes and quartiles in this graphic are irregular and *Kruskal-Wallis* test on dimensions of this box-plot has a value of 0.069. Because of the significance value is greater than 0.05, comments on this graphic may be invalid. Therefore, the results in this analysis are inconclusive.

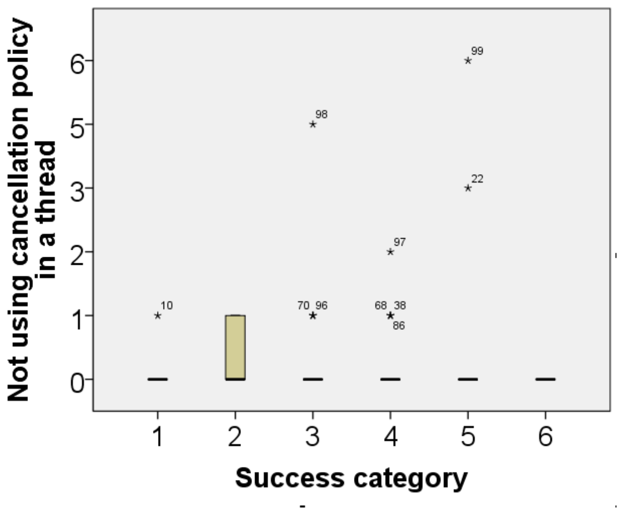


Figure 27 Box-Plot of "Not using cancellation policy in thread" count vs Success category

The boxplot in Figure 27 has a significance value which is above 0.05. With the insignificance of this box-plot graphic, this graphic cannot show any relation between "Not using cancellation policy in a thread" count and "Success category" because of the outlier values and nonoccurence of boxes.

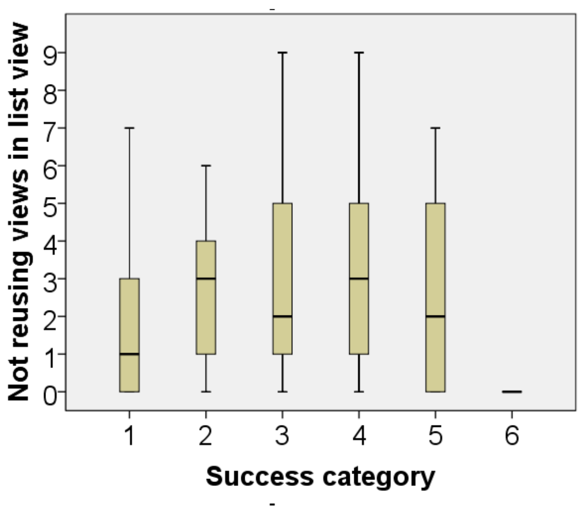


Figure 28 Box-Plot of "Not reusing views in list view" count vs Success category

Figure 28 Box-Plot of "Not reusing views in list view" count vs Success category shows the relation between "Not reusing views in list view" count and Success category. This box-plot graphic is noteworthy because it has a Kruskal-Wallis test value of 0.003,

which is below 0.05. Although the count of “Not reusing views in list view” climbs until the 3rd category which is “Bad Applications”, upper adjacent and upper hinges of 3rd, 4th, 5th and 6th categories show a falling tendency. Moreover, the applications in 6th category (very successful applications) has no “Not reusing views in list view” bad-practice. These facts show that most of the time, having “Not reusing views in list view” bad-practices in the code have an influence on the user ratings and successfulness of the applications.

5.2.3. Correlation between Average of Successfulness and Average of Bad-Practice Count

In this analysis, we looked for a relation between the average bad-practice count and the average user ratings as a successfulness metric. To achieve this goal we normalized bad-practice count of each application with the line-of-code of applications. Using the same categorization method in Table 5, we calculated the average of weighted average user ratings of applications in a category. After that, we also calculated average of the normalized bad-practice counts. Lastly, in order to examine the correlation between these two values, which are cardinal numbers, with the help of the SPSS, we computed the “Pearson product-moment correlation coefficient” values. The output of this calculation is shown in Table 7. In this table, we showed the coefficient value right across the bad-practice type. If the value of the coefficient is followed by an asterisk character, this means that the value is significant and the p-value shown on the right sight of this value is smaller than 0.05.

Table 7 Pearson product-moment correlation coefficient values between average normalized bad-practice count and average weighted average user rating (‘’ means significant correlation value which has a p-value of smaller than 0,05)*

	Pearson product-moment correlation coefficient	
Bad Practice Type	Coefficient	p-value
Using non-static inner classes	-0.973*	0.001
Not setting thread priorities	0.27	0.96
Not using cancellation policy in a thread	-0.839*	0.037
Not reusing views in list view	-0.879*	0.021

As it shown in the Table 7 “Using non-static inner classes”, “Not using cancellation policy in a thread” and “Not reusing views in list view” bad-practices has p-values smaller than 0.05 and their coefficient values are significant. For the correlation algorithm applied, they have coefficient values that are closer to -1. This implies that those three normalized bad-practices count have a strong negative relationship with the average of weighted average user ratings. In other word, having bad-practices reduces point that the user gave to the applications.

The other bad-practice in Table 7 have p-values above 0.05 for each correlation coefficient. In the light of this knowledge, the influence of this bad-practice cannot be determined by this method.

CHAPTER 6

CONCLUSION

Since user ratings gives a good hint about usefulness and success of an application, before installing application, most of the users took a glance at them. Therefore, the good reputation of the application grows exponentially by getting good ratings. We think that a good way to get good ratings from Google Play or any similar platforms is removing common bad-practices from the source code of the application. The purpose of this study is developing a tool that finds the amount of special fallacies, namely bad-practices. The other purpose is showing the relation between the amount of those bad-practices and user ratings by using the tool that finds bad-practices named Bad-Practice Finder.

Bad-Practice Finder has four bad-practice detectors: *InnerClassLeakDetector* finding inner class leaks, *ThreadPriorityNotSetDetector* finding thread priority issues, *ThreadNoCancellationPolicyDetector* finding the threads which needs cancellation policy and do not have one and *ListViewNoReuseDetector* finding list view adapter classes which do not reuse views. All of those four detectors created by using the Android Lint API explained in Section 3.1.1 and they are new and different from the bad-practices standard Android Lint checked. Android Lint API is suitable for creating new bad-practice detectors. This means that any other interested developers can contribute this work by downloading Bad-Practice-Finder.jar[21] and using it in their new projects.

Earlier experiments, for example explained in Section 3.3.1, FindBugs Google experiment shows that applying static analysis tools decreases the possibility of bugs in programs. On the other hand, Bug-Practice Finder is a static analysis tool specialized for Android development having the same conclusion and this tool is a good chance for the developers to create stable, responsive and quick applications by revealing bugs in source code. Moreover, because Bad-Practice Finder makes static analysis, unlike the explained dynamic analysis tools in Section 3.2 it has the ability to say the exact location of the problematic part of the source code. The tools in Section 3.2 can only show the existence of the problem in the application. Previous studies, explained in Section 3.4,

[21] http://www.ceng.metu.edu.tr/~e1449065/Bad-Practice-Finder.jar

looks for the security vulnerabilities, unnecessary permissions and energy bugs but they do not find the bad-practices Bad-Practice Finder detects which are the reasons behind the memory leaks, ANR dialogs.

The main contributions of this work are listed below:

- A list of bad-practices in Android Development is composed.
- Bad-Practice Finder tool, which detects bad-practices, not detected by other tools, in Android Applications, is developed.
    - Bad-practices detected by only Bad-Practice Finder:
        - Using non-static inner classes
        - Not setting thread priorities
        - Not using a cancellation policy in a thread
    - Bad-practice detected also by other tools (Section 3.1.2 PerfChecker)
        - Not reusing views in list view
- Bad-Practice Finder applied on 100 applications and this showed applicability of the tool.
- Results are analyzed with statistical methods and the relation between the user ratings and bad-practices are shown.

In this thesis we conclude that four bad-practices, which Bad-Practice Finder detects, effects the user ratings directly or indirectly. In Section 5.2, analysis indicates the negative relationship between occurrences of bad-practices and user ratings. Because those bad-practices cause crashes, freezes and slowness in the applications, this work is a good evidence that shows the users want more stable and robust applications.

First limitation in this thesis is the pool of subject applications in the experiments. We used F-Droid for open source Android applications. Google Play has over 1 million applications but we were able to find approximately 1200 applications from the F-Droid. For example, although an application has very low rates and users are complains about the application, if it is not one of the 1200 open source applications in F-Droid, which is our open source application source it is not possible for us to analyze this application with our Bad-Practice Finder. This limitation may affect correct representation of applications in our experiments. Other than this, some users may not give objective ratings to an application on Google Play because of the psychological or sociological reasons and this may cause some noisy or incorrect data in our experiments. In addition, our tool, Bad Practice Finder, searches for some bad-practices that can be found only heuristic approaches as indicated in Section 4.2.3. Last limitation blurring our statistical findings is the fact that there may be factors that reduce the user ratings other than bad-practices analyzed in this study. For example, the reason behind the low user ratings may be the content of an application or user-unfriendly user interfaces.

In the future, we plan to add new bad-practices to our list and detect these practices in Bad-Practice Finder. In addition, we will do our experiments and analysis on a bigger application set. Lastly, we plan to examine the applications in the Google Play that users most complain against the bad-practices.

REFERENCES

[1] "Android Design Principles," Android Open Source Project, [Online]. Available: http://developer.android.com/design/get-started/principles.html. [Accessed 15 07 2014].

[2] Android Open Source Project, "Core App Quality Guidelines," [Online]. Available: http://developer.android.com/distribute/googleplay/quality/core.html. [Accessed 1 12 2013].

[3] J. Brutlag, "Speed Matters," 24 06 2009. [Online]. Available: http://googleresearch.blogspot.com/2009/06/speed-matters.html. [Accessed 25 12 2013].

[4] Android Open Source Project, "Keeping Your App Responsive," [Online]. Available: http://developer.android.com/training/articles/perf-anr.html. [Accessed 1 12 2013].

[5] M. Burton and D. Felker, Android Application Development for Dummies, Wiley, 2012.

[6] R. Meier, Professional Android 2 Application Development, Wrox, 2010 .

[7] "Creative Bloq," [Online]. Available: http://www.creativebloq.com/app-design/how-build-app-tutorials-12121473. [Accessed 11 11 2013].

[8] C. Smith, "Free online Android programming course starting next month," Android Authority Beta, 27 December 2013 . [Online]. Available: http://www.androidauthority.com/free-online-android-programming-course-

327826/. [Accessed 18 01 2013].

[9] R. Meier, "Android operating system," in *Professional Android 4 Application Development*, 2012 , pp. 1-19.

[10] "Google Play," Wikimedia Foundation, Inc, 23 06 2014. [Online]. Available: http://en.wikipedia.org/wiki/Google_Play.

[11] F. Ableson, "Introduction to Android development," IBM, 12 05 2009. [Online]. Available: https://www.ibm.com/developerworks/library/os-android-devel/. [Accessed 25 06 2014].

[12] "Activity," Android Open Source Project, 20 06 2014. [Online]. Available: http://developer.android.com/reference/android/app/Activity.html.

[13] "Managing the Activity Lifecycle," Android Open Source Project., [Online]. Available: http://developer.android.com/training/basics/activity-lifecycle/index.html. [Accessed 25 06 2014].

[14] "Context," Android Open Source Project., [Online]. Available: http://developer.android.com/reference/android/content/Context.html. [Accessed 25 06 2014].

[15] "Services," Android Open Source Project, [Online]. Available: http://developer.android.com/guide/components/services.html. [Accessed 25 06 2014].

[16] "Intents and Intent Filters," Android Open Source Project, [Online]. Available: http://developer.android.com/guide/components/intents-filters.html. [Accessed 25 06 2014].

[17] "Content Providers," Android Open Source Project., [Online]. Available: http://developer.android.com/guide/topics/providers/content-providers.html. [Accessed 25 06 2014].

[18] "Android Tools Project Site," [Online]. Available: http://tools.android.com/tips/lint. [Accessed 02 12 2013].

[19] "Android Tools Project Site," [Online]. Available: http://tools.android.com/tips/lint/writing-a-lint-check. [Accessed 07 11

2013].

[20] Y. Liu and C. Xu , "VeriDroid: automating Android application verification," in *MDS '13 Proceedings of the 2013 Middleware Doctoral Symposium*, New York, NY, USA, 2013 .

[21] Y. Liu, C. Xu and S.-C. Cheung, "Characterizing and Detecting Performance Bugs for Smartphone Applications," in *ICSE, the International Conference on Software Engineering*, Hyderabad, India , 2014.

[22] Android Open Source Project, "UI/Application Exerciser Monkey," [Online]. Available: http://developer.android.com/tools/help/monkey.html. [Accessed 25 12 2013].

[23] D. Hovemeyer and W. Pugh, "Finding Bugs is Easy," in *OOPSLA 2004*, Vancouver, BC, Canada, 2004.

[24] N. Ayewah and W. Pugh, "The Google FindBugs Fixit," in *ISSTA'10*, Trento, 2010.

[25] D. H. Hovemeyer and W. W. Pugh, "FindBugs Bug Descriptions," University of Maryland, 23 11 2013. [Online]. Available: http://findbugs.sourceforge.net/bugDescriptions.html.

[26] D. H. Hovemeyer and W. W. Pugh, "FindBugs™ - Find Bugs in Java Programs," University of Maryland , 22 10 2013. [Online]. Available: http://findbugs.sourceforge.net/index.html.

[27] K. Lee, J. Flinn, T. Giuli, B. Noble and C. Peplin, "AMC: verifying user interface properties for vehicular applications," in *MobiSys '13*, New York, 2013.

[28] E. Chin, A. P. Felt, K. Greenwood and D. Wagner, "Analyzing inter-application communication in Android," in *MobiSys '11*, New York, 2011.

[29] A. Pathak, A. Jindal, Y. C. Hu and S. P. Midkiff, "What is keeping my phone awake?: characterizing and detecting no-sleep energy bugs in smartphone apps," in *MobiSys '12*, New York, 2012.

[30] A. P. Felt, E. Chin, S. Hanna, D. Song and D. Wagner, "Android Permissions Demystified," in *CCS 2011*, Chicago, 2011.

[31] Cunningham & Cunningham, Inc., "Inner Class," 16 4 2010. [Online]. Available: http://c2.com/cgi/wiki?InnerClass. [Accessed 05 12 2013].

[32] R. Guy, "Avoiding memory leaks," 19 January 2009. [Online]. Available: http://android-developers.blogspot.com/2009/01/avoiding-memory-leaks.html. [Accessed 01 12 2013].

[33] R. Guy, "Android Developers Blog," 06 05 2009. [Online]. Available: http://android-developers.blogspot.com/2009/05/painless-threading.html.

[34] Compuware, "Java Enterprise Performance Book (dynatrace)," [Online]. Available: http://javabook.compuware.com/content/memory/how-garbage-collection-works.aspx. [Accessed 2012].

[35] A. Lockwood, "Android Design Patterns," 15 04 2013. [Online]. Available: http://www.androiddesignpatterns.com/2013/04/activitys-threads-memory-leaks.html. [Accessed 12 11 2013].

[36] L. Rocha, "Lucas Rocha Blog," 05 04 2012. [Online]. Available: http://lucasr.org/2012/04/05/performance-tips-for-androids-listview/.

[37] "Android Tools Project Site," [Online]. Available: http://tools.android.com/tips/lint-checks. [Accessed 25 12 2013].

[38] M. Sipser, "The Halting Problem," in *Introduction to the Theory of Computation (Second Edition ed.)*, Boston, Thomson Course Technology, 2006, pp. 179-181.

[39] F-Droid Limited, "Category:Apps," F-Droid, 06 09 2013. [Online]. Available: https://f-droid.org/wiki/index.php?title=Category:Apps.

[40] "Selenium," [Online]. Available: http://docs.seleniumhq.org/.

[41] N. J. Cox, "The Stata Journal," *Speaking Stata: Creating and varying box plots,* no. Box plots, p. 478–496, 19 07 2014.

APPENDIX A

WEBSITES EXPLAINING BAD AND BEST PRACTICES

Table 8 Websites explaining bad and best practices

Title	Link
Displaying Bitmaps Efficiently	http://developer.android.com/training/displaying-bitmaps/index.html
Activities, Threads, & Memory Leaks	http://www.androiddesignpatterns.com/2013/04/activitys-threads-memory-leaks.html#more
How to Leak a Context: Handlers & Inner Classes	http://www.androiddesignpatterns.com/2013/01/inner-class-handler-memory-leak.html
Handling Configuration Changes with Fragments	http://www.androiddesignpatterns.com/2013/04/retaining-objects-across-config-changes.html#more
Avoiding memory leaks	http://android-developers.blogspot.in/2009/01/avoiding-memory-leaks.html
Is bad-practice to use Context as static variable?	http://stackoverflow.com/questions/17691656/bad-practice-to-use-context-as-static-variable
Avoid Creating Unnecessary Objects	http://developer.android.com/training/articles/perf-tips.html#ObjectCreation
Avoid Internal Getters/Setters	http://developer.android.com/training/articles/perf-tips.html#GettersSetters
Use Enhanced For Loop Syntax	http://developer.android.com/training/articles/perf-tips.html#Loops
Implement the run() method	http://developer.android.com/training/multiple-threads/define-runnable.html#RunMethod
Hold View Objects in a View Holder	http://developer.android.com/training/improving-layouts/smooth-scrolling.html#ViewHolder
Google I/O 2009 - Make your Android UI Fast and Efficient	http://www.youtube.com/watch?v=N6YdwzAvwOA&feature=player_detailpage#t=1199

Title	Link
Memory Leak issue while excuting a piece of Code in Android	http://stackoverflow.com/questions/18525453/memory-leak-issue-while-excuting-a-piece-of-code-in-android
Core App Quality Guidelines	http://developer.android.com/distribute/googleplay/quality/core.html
Android Best Practices and Tips	http://clayallsopp.com/posts/android-best-practices-tips/
Android Performance Case Study	http://www.curious-creature.org/2012/12/01/android-performance-case-study/
What are some best practices in Android memory management?	http://www.quora.com/Android-Development/What-are-some-best-practices-in-Android-memory-management
What Tips Would You Give to Someone Who is Just Starting Out Developing Apps?	http://www.instantshift.com/2013/08/08/app-development-tips-for-beginners/
Ruleset Android	http://pmd.sourceforge.net/pmd-5.0.5/rules/java/android.html
13 must-have features for your next mobile app	http://thenextweb.com/entrepreneur/2012/12/16/13-must-have-features-for-your-business-mobile-app/14/
Google I/O 2010 - Writing zippy Android apps	http://www.youtube.com/watch?v=c4znvD-7VDA
Keeping Your App Responsive	http://developer.android.com/training/articles/perf-anr.html
Android Developers Blog	http://android-developers.blogspot.com/2009/05/painless-threading.html
Lucas Rocha Blog	http://lucasr.org/2012/04/05/performance-tips-for-androids-listview/
Lint checks	http://tools.android.com/tips/lint-checks
Android Internals and Bad-Practices	http://s3.amazonaws.com/ppt-download/codestrong2012breakoutsession-androidinternalsandbestpractices-121106145722-phpapp01.pptx
Turbo-charge your UI	http://dl.google.com/io/2009/pres/Th_0230_TurboChargeYourUI-HowtomakeyourAndroidUIfastandefficient.pdf
Android Best Practices	http://www.slideshare.net/harala/android-best-practices
Best Practices for Mobile Application Development on Android	http://www.slideshare.net/tasneemsayeed/best-practices-for-mobile-app-development-android-march-15-2013-ts

Title	Link
Common Java problems when coding for Android and advice for dealing with them	http://www.slideshare.net/sgilmore/common-java-problems-when-developing-with-android

APPENDIX B

BAD PRACTICES LIST

Table 9 Bad-Practices List

Bad-Practices	Occurrence Rate (%)	Problem
Passing activity context	38,78	Passing activity context to out of its scope classes may create memory leaks because activity lifetime cannot be determined.
Lack of cancellation policy in threads	14,98	Explained in Section 4.1.3
Not handling configuration changes	7,83	Normally, Android Operating System destroys and recreates the current activity when the device orientation changes. However using potentially large objects, such as WebMaps or Layers, on activity creation do not supported by mechanism used by the Android Operating System. Not handling this type of situations creates problems with memory and user experiences.
Not checking for API availability before using features	7,41	Devices have many features such as GPS, some of the devices may not have these features and using these features without controlling the existence makes null point de-references.
Using non-static inner classes	6,81	Explained in Section 4.1.1

Using too much nested layouts	6,29	Using too much nested layouts creates "Out of Memory Exception"'s. Useless layouts must be avoided.
Not reusing list view items.	4,05	Explained in Section 4.1.4
Not setting priority of thread in "run" method	2,27	Explained in Section 4.1.2
Loading too many big images	3,91	Loading too many big images creates "Out of Memory Exceptions"
Permission mistakes	2,15	In general, forgetting to add permissions of a task make the call of that task impossible and the application crashes with permission exception.
Keeping static drawables or view	2,1	Undestroyed drawables creates references and these references prevents Garbage Collector to collect navigated activities. This situation creates memory leaks
Potentially long running and not threaded operations like network, db etc. in main thread	1,76	Potentially long running operations blocks main thread and may not allow main thread answer in 5 seconds. This causes ANR dialogs displayed
Not allowing the runtime to kill your service	1,26	Services starts automatically by default if Service.START_NOT_STICKY is not used. Starting service every time it is killed may not be necessary and it may use a lot unnecessary resources.
Hardcoding file locations	0,4	Using hard coded file references may lead to null point dereferences

APPENDIX C

APPLICATIONS WITH RATINGS, INSTALL COUNTS AND LINE OF CODES

Table 10 Applications with ratings install counts and line of codes

Application Id	Using non-static inner classes	Not setting thread priorities	Not using a cancellation policy in a thread	Not reusing views in list view	5 point count	4 point count	3 point count	2 point count	1 point count	Weighted average	Install count	Line of code
am.ed.importcontacts	12	0	0	0	9	0	3	0	9	3	1000	3717
at.bitfire.davdroid	0	0	0	1	67	15	6	3	18	4	5000	4639
at.univie.sensorium	16	9	0	0	10	0	0	0	0	5	500	3942
ch.nexuscomputing.android.osciprimeics	16	18	3	0	20	4	0	2	1	4,5	5000	7036
com.aripuca.tracker	3	9	0	0	18	6	4	2	3	4	5000	8559
com.axelby.podax	17	10	1	3	32	27	15	10	11	3,6	5000	10115
com.btmura.android.reddit	15	12	0	1	50	29	5	4	8	4,1	10000	16500
com.codebutler.farebot	0	1	0	2	258	104	77	71	116	3,5	100000	27665
com.coinbase.android	26	12	0	5	1638	546	159	79	132	4,4	500000	9259
com.commonsware.andr	2	18	0	0	290	76	31	2	19	4,5	50000	3571

oid.arXiv												
com.dozuki.ifixit	11	7	0	3	2167	717	314	126	214	4,3	1000000	17731
com.duckduckgo.mobile.android	17	5	0	6	5137	1294	424	195	238	4,5	500000	10660
com.episode6.android.appalarm.pro	8	16	1	1	392	107	45	45	128	3,8	100000	4414
com.euedge.openaviationmap.android	4	5	0	0	18	3	4	3	6	3,7	10000	3186
com.frostwire.android	15	46	5	3	20078	5017	3009	1592	5059	4	5000000	314169
com.gcstar.viewer	14	1	1	0	28	12	10	7	18	3,3	5000	3631
com.gmail.charleszq	17	4	0	1	38	25	11	2	5	4,1	50000	8652
com.googamaphone.typeandspeak	8	1	0	1	3104	1047	657	330	782	3,9	1000000	2366
com.googlecode.gtalksms	35	12	0	0	301	68	24	14	34	4,3	100000	37370
com.gpl.rpg.AndorsTrail	70	2	0	0	5829	2028	929	427	574	4,2	1000000	16865
com.hectorone.multismssender	4	0	0	3	107	50	21	11	43	3,7	50000	1588
com.hobbyone.HashDroid	0	2	0	0	448	67	10	5	9	4,7	50000	4317
com.liato.bankdroid	21	7	0	2	2829	993	280	129	256	4,3	500000	19470
com.lightbox.android.camera	15	7	0	3	448	170	90	25	57	4,2	500000	9236
com.maxfierke.sandwichroulette	2	1	0	0	10	0	0	0	0	5	1000	1443
com.morphoss.acal	54	10	0	7	21	19	12	10	31	2,9	5000	31436
com.mykola.lexinproject	10	9	0	0	121	25	13	8	25	4,1	50000	2845
com.nanoconverter.zlab	8	0	0	0	112	9	4	1	4	4,7	5000	1649
com.newsblur	11	5	0	1	412	248	114	61	174	3,7	100000	30927
com.nilhcem.frcndict	6	1	0	1	51	31	13	6	16	3,8	50000	3668
com.nolanlawson.keepscore	5	24	0	7	425	98	24	11	18	4,6	50000	10813
com.nolanlawson.logcat	24	14	0	5	2000	491	166	58	86	4,5	500000	5642
com.owncloud.android	13	2	0	6	862	728	471	256	328	3,6	100000	40274

com.piratebayfree	2	2	0	4	2547	806	346	204	376	4,2	500000	3252
com.qubling.sidekick	20	10	0	0	2	8	0	0	2	3,7	500	4211
com.rhiannonweb.android.migrainetracker	0	0	0	1	3	0	2	1	6	2,4	5000	505
com.ruesga.android.wallpapers.photophase	19	10	0	2	55	16	5	4	9	4,2	10000	15391
com.sam.hex	7	45	0	1	402	206	112	54	96	3,9	50000	6948
com.teamdc.stephendiniz.autoaway	4	2	0	2	9	3	0	2	2	3,9	5000	2950
com.teleca.jamendo	32	4	0	1	810	361	146	106	247	3,8	500000	8709
com.tkjelectronics.balanduino	6	14	0	0	9	1	0	0	0	4,9	500	11290
com.valleytg.oasvn.android	4	17	0	2	20	12	5	4	4	3,9	5000	4832
com.zachrattner.pockettalk	0	0	0	0	11	0	0	0	0	5	1000	836
com.zapta.apps.maniana	39	14	0	0	229	47	16	5	7	4,6	50000	14448
de.blau.android	31	3	0	8	162	100	57	23	35	3,9	50000	11741
de.jdsoft.law	9	0	0	1	15	0	0	0	5	4	1000	41041
de.jurihock.voicesmith	22	5	0	3	38	14	16	6	21	3,4	50000	10183
de.koelle.christian.trickytripper	3	6	0	4	48	10	2	1	2	4,6	5000	38747
de.mbutscher.wikiandpad.alphabeta	24	12	0	3	16	8	0	1	1	4,4	5000	10509
de.onyxbits.textfiction	0	0	1	0	272	134	65	23	36	4,1	50000	6451
de.skubware.opentraining	9	2	0	4	25	11	12	8	9	3,5	10000	8127
dk.andsen.asqlitemanager	2	2	0	0	813	262	83	29	37	4,5	500000	8078
dk.nindroid.rss	15	10	2	3	6935	2769	1069	326	547	4,3	5000000	19768
es.cesar.quitesleep	44	15	1	3	44	9	7	3	24	3,5	50000	26001
eu.hydrologis.geopaparazzi	29	10	0	6	80	10	4	2	7	4,5	10000	17782

info.guardianproject.gpg	22	8	0	1	50	13	16	10	15	3,7	50000	31297
it.sasabz.android.sasabus	2	0	0	3	51	29	21	13	29	3,4	50000	8908
kr.hybdms.sidepanel	51	16	1	4	47	12	18	9	19	3,6	10000	22056
li.klass.fhem	26	11	0	6	213	32	6	2	4	4,7	50000	19895
mobi.cyann.nstools	4	0	0	0	1251	105	23	11	21	4,8	500000	3257
nerd.tuxmobil.fahrplan.congress	2	3	0	0	187	28	6	0	1	4,8	10000	4231
net.cyclestreets	12	0	0	6	148	79	22	19	29	4	100000	13686
net.dahanne.android.regalandroid	4	1	0	1	39	37	35	21	35	3,1	10000	9137
net.debian.debiandroid	9	0	0	0	41	5	3	0	0	4,8	5000	30047
net.gaast.giggity	28	3	0	5	30	10	4	2	4	4,2	5000	4736
net.sf.times	6	4	0	3	194	41	17	12	15	4,4	50000	8260
net.sourceforge.subsonic.androidapp	25	26	0	3	3910	939	269	163	417	4,4	500000	10789
org.ale.openwatch	2	8	0	1	75	21	13	10	55	3,3	50000	1608
org.dmfs.tasks	2	2	0	1	300	154	72	28	24	4,2	100000	7893
org.dolphinemu.dolphinemu	5	0	0	0	4939	1154	1193	872	2826	3,4	1000000	1812
org.droidupnp	14	9	1	1	18	11	8	4	5	3,7	10000	5594
org.eehouse.android.xw4	46	35	0	4	118	62	41	34	52	3,5	100000	19514
org.fedorahosted.freeotp	6	3	0	0	190	28	4	2	2	4,8	5000	6107
org.geometerplus.zlibrary.ui.android	51	98	0	7	71860	11347	3129	1388	3098	4,6	10000000	53793
org.gmote.client.android	15	7	6	1	17481	4332	1205	437	898	4,5	5000000	27257
org.gnucash.android	1	3	0	2	326	177	100	50	59	3,9	50000	6135
org.ironrabbit.bhoboard	13	119	0	9	268	63	50	29	48	4	50000	77105
org.jtb.httpmon	0	3	0	2	83	36	15	11	12	4,1	50000	3936
org.liberty.android.fantastischmemo	52	32	0	2	1132	284	79	40	60	4,5	500000	24095
org.mult.daap	20	6	0	5	298	175	69	26	68	4	500000	10313

org.musicbrainz.picard.barcodescanner	0	0	0	0	7	2	1	0	0	4,6	5000	812
org.nick.wwwjdic	36	60	0	9	635	221	85	45	50	4,3	500000	18982
org.npr.android.news	39	23	1	7	9286	4427	1867	1140	1450	4	5000000	13075
org.pyload.android.client	35	28	0	2	296	92	29	7	19	4,4	50000	64609
org.qii.weiciyuan	65	5	0	3	2583	551	107	28	26	4,7	50000	24653
org.scid.android	24	15	0	2	260	70	22	8	13	4,5	50000	12803
org.servalproject	42	15	1	3	667	120	76	52	89	4,2	100000	42949
org.sickstache	13	0	0	2	146	67	7	4	7	4,5	50000	11784
org.thialfihar.android.apg	19	4	0	4	1450	511	146	56	89	4,4	500000	97950
org.totschnig.myexpenses	8	0	0	9	960	446	138	53	83	4,3	500000	14737
org.wheelmap.android.online	9	2	0	5	98	40	17	6	8	4,3	50000	8888
org.wordpress.android	47	41	0	5	23404	10651	4228	1601	2599	4,2	5000000	33611
org.yaaic	56	18	1	6	818	336	139	57	87	4,2	500000	30120
org.zakky.memopad	0	0	0	0	9	3	3	0	0	4,4	1000	641
org.zeitgeist.movement	22	4	0	0	340	19	8	2	7	4,8	50000	6241
org.zirco	13	13	0	5	132	89	37	15	44	3,8	100000	9695
pl.nkg.geokrety	6	0	0	0	36	10	2	0	3	4,5	1000	4545
ru.valle.btc	8	5	0	0	22	5	0	0	2	4,6	5000	3791
uk.org.cardboardbox.wonderdroid	4	1	0	2	327	87	78	37	92	3,8	100000	1766
uk.org.ngo.squeezer	38	19	1	3	112	64	24	13	17	4	50000	9824

APPENDIX D

CLASS DIAGRAMS

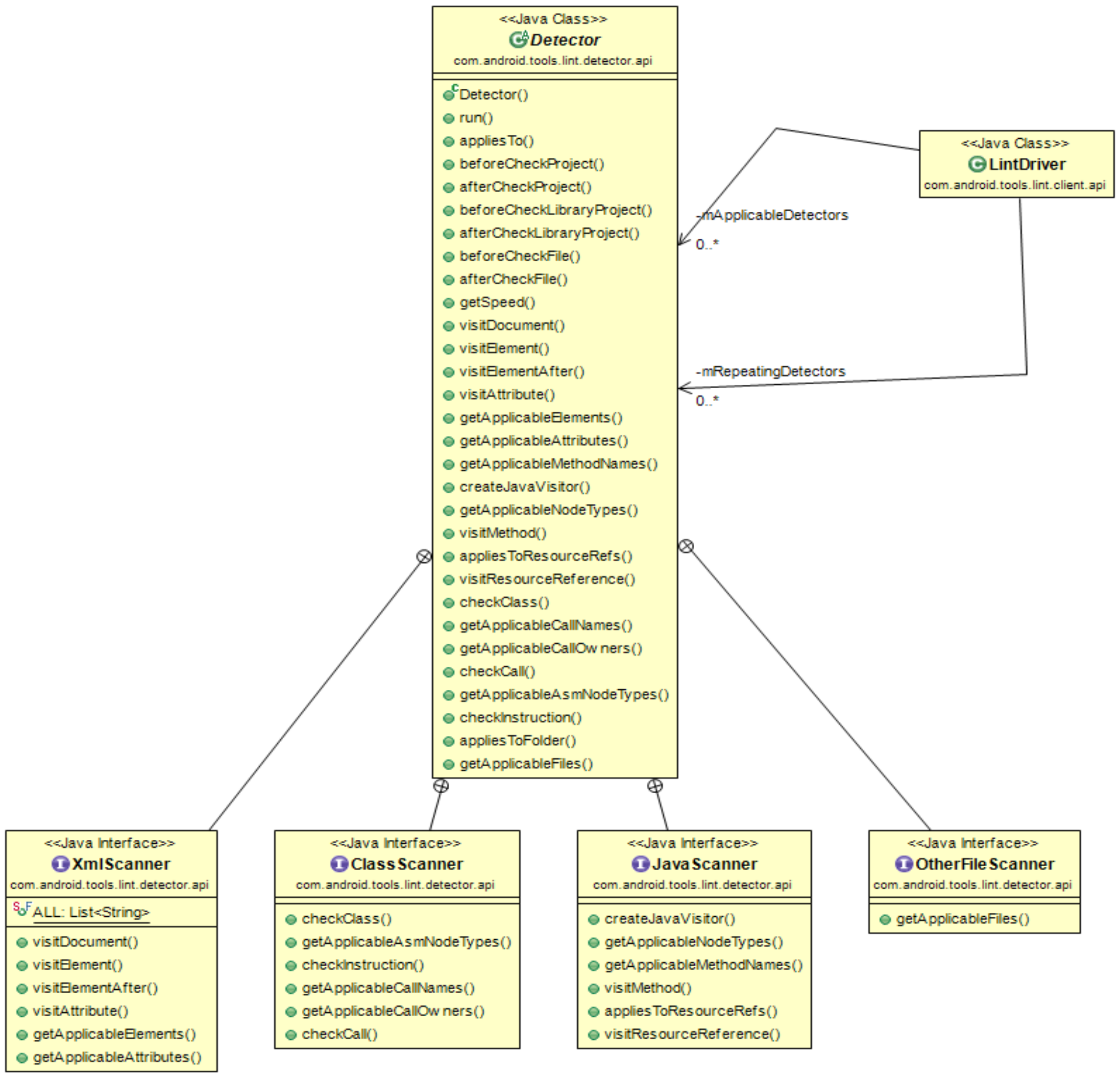
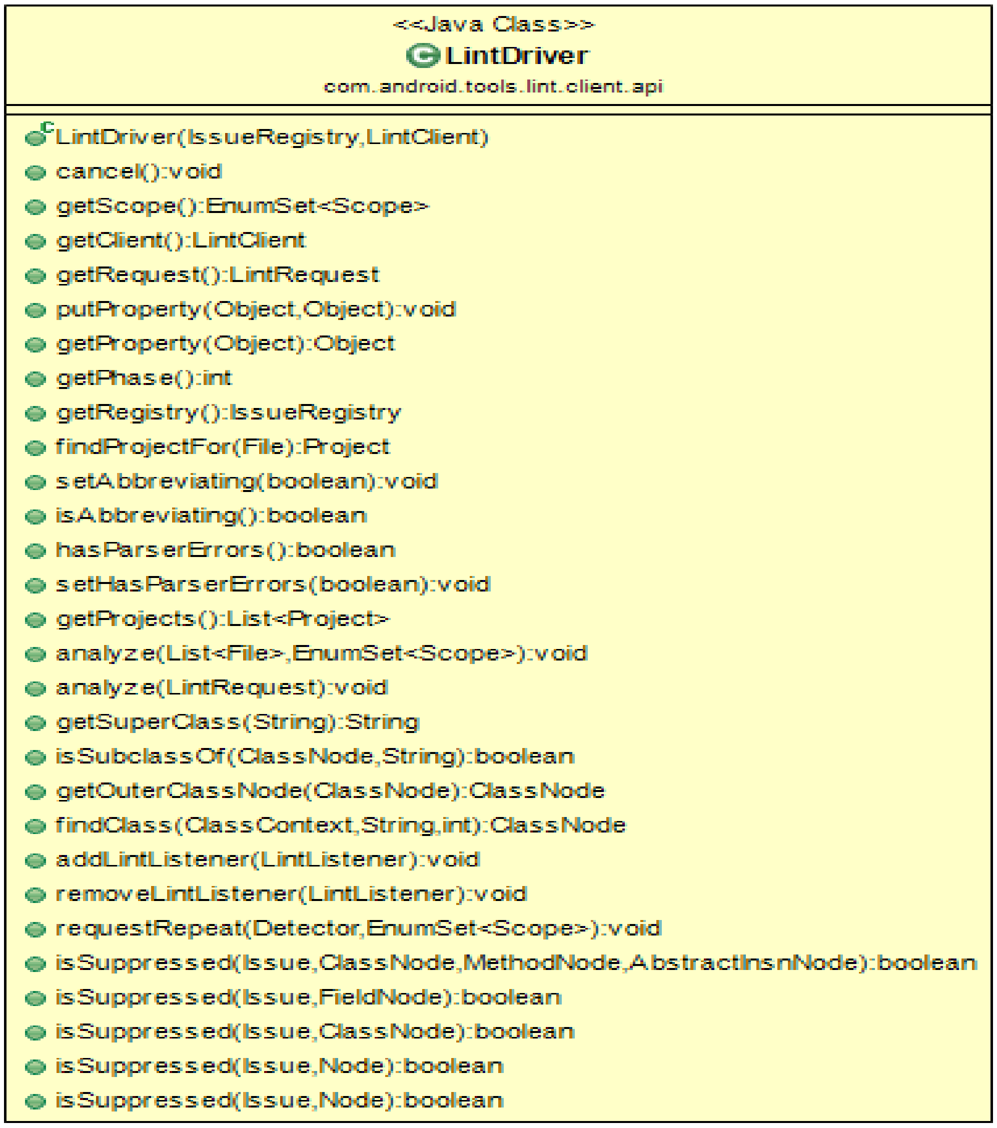


Figure 29 Detector class diagram



<<Java Class>>
LintDriver
com.android.tools.lint.client.api
LintDriver(IssueRegistry,LintClient)
cancel():void
getScope():EnumSet<Scope>
getClient():LintClient
getRequest():LintRequest
putProperty(Object,Object):void
getProperty(Object):Object
getPhase():int
getRegistry():IssueRegistry
findProjectFor(File):Project
setAbbreviating(boolean):void
isAbbreviating():boolean
hasParserErrors():boolean
setHasParserErrors(boolean):void
getProjects():List<Project>
analyze(List<File>,EnumSet<Scope>):void
analyze(LintRequest):void
getSuperClass(String):String
isSubclassOf(ClassNode,String):boolean
getOuterClassNode(ClassNode):ClassNode
findClass(ClassContext,String,int):ClassNode
addLintListener(LintListener):void
removeLintListener(LintListener):void
requestRepeat(Detector,EnumSet<Scope>):void
isSuppressed(Issue,ClassNode,MethodNode,AbstractInsnNode):boolean
isSuppressed(Issue,FieldNode):boolean
isSuppressed(Issue,ClassNode):boolean
isSuppressed(Issue,Node):boolean
isSuppressed(Issue,Node):boolean

Figure 30 Lint Driver class diagram

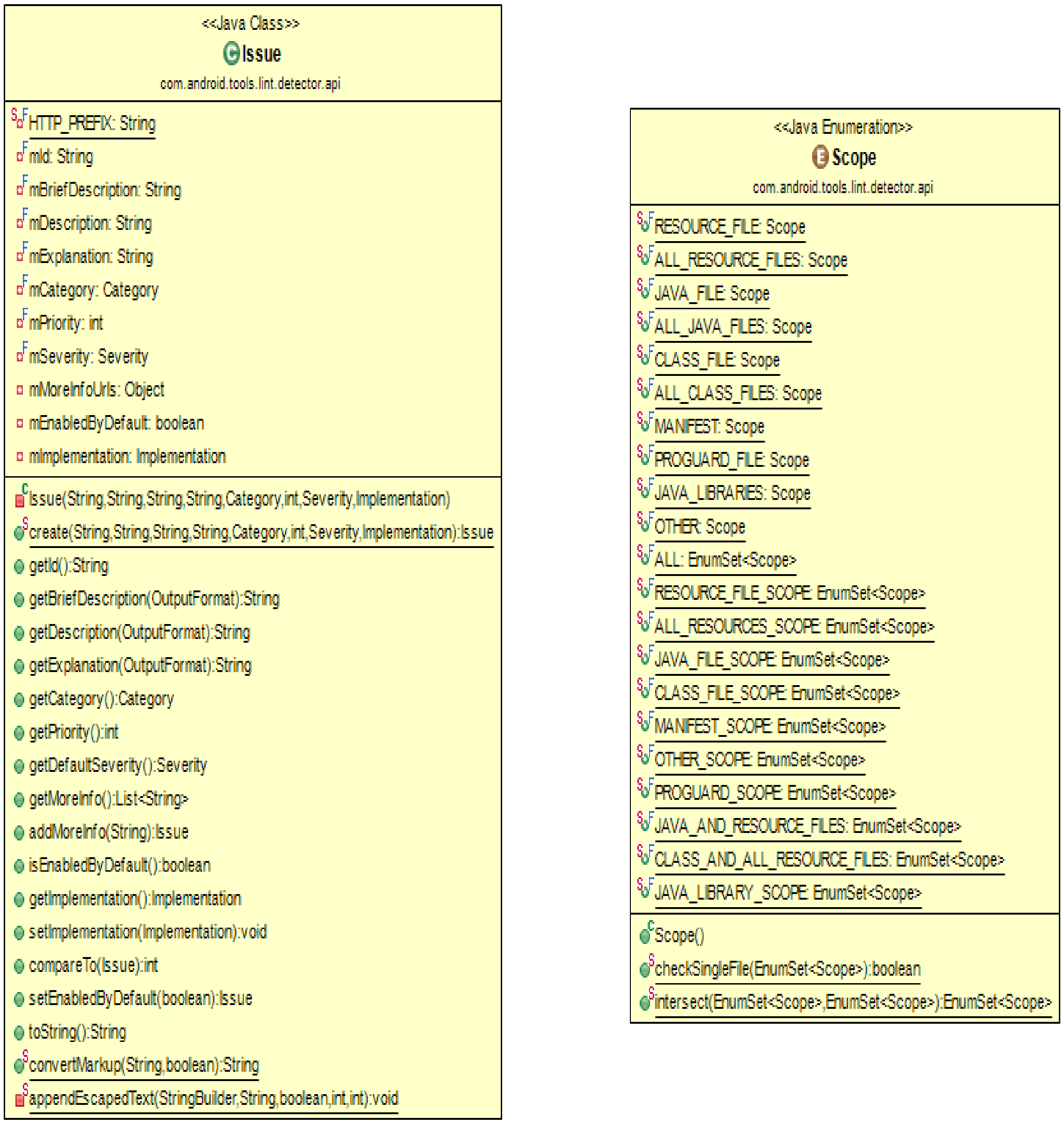

<<Java Class>>
Issue
com.android.tools.lint.detector.api
HTTP_PREFIX: String
mId: String
mBriefDescription: String
mDescription: String
mExplanation: String
mCategory: Category
mPriority: int
mSeverity: Severity
mMoreInfoUrls: Object
mEnabledByDefault: boolean
mImplementation: Implementation
Issue(String,String,String,String,Category,int,Severity,Implementation)
create(String,String,String,String,Category,int,Severity,Implementation):Issue
getId():String
getBriefDescription(OutputFormat):String
getDescription(OutputFormat):String
getExplanation(OutputFormat):String
getCategory():Category
getPriority():int
getDefaultSeverity():Severity
getMoreInfo():List<String>
addMoreInfo(String):Issue
isEnabledByDefault():boolean
getImplementation():Implementation
setImplementation(Implementation):void
compareTo(Issue):int
setEnabledByDefault(boolean):Issue
toString():String
convertMarkup(String,boolean):String
appendEscapedText(StringBuilder,String,boolean,int,int):void
<<Java Enumeration>>
Scope
com.android.tools.lint.detector.api
RESOURCE_FILE: Scope
ALL_RESOURCE_FILES: Scope
JAVA_FILE: Scope
ALL_JAVA_FILES: Scope
CLASS_FILE: Scope
ALL_CLASS_FILES: Scope
MANIFEST: Scope
PROGUARD_FILE: Scope
JAVA_LIBRARIES: Scope
OTHER: Scope
ALL: EnumSet<Scope>
RESOURCE_FILE_SCOPE: EnumSet<Scope>
ALL_RESOURCES_SCOPE: EnumSet<Scope>
JAVA_FILE_SCOPE: EnumSet<Scope>
CLASS_FILE_SCOPE: EnumSet<Scope>
MANIFEST_SCOPE: EnumSet<Scope>
OTHER_SCOPE: EnumSet<Scope>
PROGUARD_SCOPE: EnumSet<Scope>
JAVA_AND_RESOURCE_FILES: EnumSet<Scope>
CLASS_AND_ALL_RESOURCE_FILES: EnumSet<Scope>
JAVA_LIBRARY_SCOPE: EnumSet<Scope>
Scope()
checkSingleFile(EnumSet<Scope>):boolean
intersect(EnumSet<Scope>,EnumSet<Scope>):EnumSet<Scope>

Figure 31 Scope & issue class diagram

TEZ FOTOKOPİSİ İZİN FORMU

ENSTİTÜ

Fen Bilimleri Enstitüsü	☐
Sosyal Bilimler Enstitüsü	☐
Uygulamalı Matematik Enstitüsü	☐
Enformatik Enstitüsü	☒
Deniz Bilimleri Enstitüsü	☐

YAZARIN

Soyadı : Sağlam
Adı : İsmail Alper
Bölümü : Bilişim Sistemleri (Information Systems)

TEZİN ADI (İngilizce) : Measuring and Assesment of Well Known Bad Practices in Android Applications

TEZİN TÜRÜ : Yüksek Lisans ☒ Doktora ☐

1. Tezimin tamamından kaynak gösterilmek şartıyla fotokopi alınabilir. ☒
2. Tezimin içindekiler sayfası, özet, indeks sayfalarından ve/veya bir bölümünden kaynak gösterilmek şartıyla fotokopi alınabilir. ☐
3. Tezimden bir (1) yıl süreyle fotokopi alınamaz. ☐

TEZİN KÜTÜPHANEYE TESLİM TARİHİ : …………………….