12-2018

Bad Droid! An in-depth empirical study on the occurrence and impact of Android specific code smells

Khalid Saeed Almalki
ksa8566@rit.edu

Bad Droid! An in-depth empirical study on the occurrence and impact of Android specific code smells

by

Khalid Saeed Almalki

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of
Master of Science in Software Engineering

Supervised by
Dr. Mohamed Wiem Mkaouer

Department of Software Engineering
B. Thomas Golisano College of Computing and Information Sciences
Rochester Institute of Technology
Rochester, New York

December 2018

The thesis "Bad Droid! An in-depth empirical study on the occurrence and impact of Android specific code smells" by Khalid Saeed Almalki has been examined and approved by the following Examination Committee:

Dr. Mohamed Wiem Mkaouer
Assistant Professor
Thesis Committee Chair

Dr. Yasmine El-Glaly
Lecturer

Dr. Christian D. Newman
Assistant Professor

Dr. J. Scott Hawker
Associate Professor
Graduate Program Director

Acknowledgments

I would like to thank everyone who supported me to accomplish this mighty task! I am grateful to my advisor Mohamed Wiem Mkaouer for his encouragement and advice during my research journey. I am also grateful to faculty and staff of the Software Engineering Department and English Language Center for all the knowledge and support extended to me during my time at RIT.

Special thanks to my family and friends for all your patience and support. Without your encouragement, I would not be the person I am today.

Abstract

Knowing the impact of bad programming practices or code smells has led researchers to conduct numerous studies in software maintenance. Most of the studies have defined code smells as bad practices that may affect the quality of the software. However, most of the existing research is heavily focused on detecting traditional code smells and less focused on mobile application specific Android code smells. Presently, there is a few papers that focus on android code smells - a catalog for Android code smells. This catalog defines 30 Android specific code smell that may impact maintainability of an app. In this research, we plan to introduce a detector tool called *BadDroidDetector* for Android code smells that can detect 13 code smells from the catalog. We will also conduct an empirical study to know the distribution of 13 smell that we detect and know the severity of these smells.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Code smells [8] identify as bad design or coding practices. Code smells are not the same as bugs and do not mean that the code deviates from the expected execution, rather than design rules were violated, which may lead to long-term maintainability problems and technical debt. On other researchers, Code smell defined as poor or bad practices that impact the software maintainability negatively.

As of recent times, the popularity of mobile applications has seen an increase, which means developers see an increase in market share. While there a huge number of applications, some of the developers try to write code as fast as they can without consideration of the code quality, especially code smells; this results in issues with the maintainability of the system.

According to [19], around 32 of the developers are not aware of code smells and their pitfalls. As Ahmed et al.[15] claimed that there is a gap between the code smells that receive a lot of attention in the research and those that appear n real-world applications, we will try to build tool to capture the missing code smell which will simulate the real-life code smell.

After analyzing the papers that have conducted a study on Android code smell, we found that few researchers are focusing on mobile applications. In this context, we found [11] proposed a set of android code smell specific for Android. These Android-specific smells may threat several non-functional attributes of mobile apps, such as data integrity, and source code quality. As highlighted by Hetch et al. [9], these type of smells can also lead to performance issues. Due to the limited resource of mobile devices including the

CPU and memory, it is essential to have a such a way to detect the code smells that may affect the effectiveness and usability of using mobile applications. [11] has proposed a list of the code smells that are associated with the problem.

Studying the available tool that detects the Android code smell, we found two tools can identify a limited number of Android code smells. The first tool is aDoctor was built by [16] to detect a limited number of code smells proposed by [11]. The tool utilizes the Abstract Syntax Tree of the source code and applies detection rules based on the definitions of the smells provided by Reimann et al. [11]. The second tool is Paprika [10] that can detect software anti-patterns and also four specific code smell for android. In other words, the tool can detect OO code smell, and even 4 type of android code smells which are Member Ignoring Method (MIM), Leaking Inner Class (LIC), UI Overdraw (UIO) and Heavy Broadcast Receiver (HBR).

Due to the limitation of tools to detect all android code smells defined by Reimann et al. [11] and high growth of developing an Android application which developer build the application on a limited resource (the CPU and memory). It is important to build apps have fewer quality issues since there is a list of Android code smell that should be avoided. In 2013, Google Play store contained almost 2 millions of Android apps in the market of which 11% are classified as low-quality apps. Due to the aforementioned reason, we aim to extend the aDoctor tool to detect 13 code smells, defined by Reimann et al. [11]. In other words, we will introduce a new code smell detector called *BadDroidDetector* that identifies 13 Android-specific code smells which that have not been in prior detector tools. The tool will use the Reimann code smell rules to apply them in the detection strategy. Also, we will perform a large-scale empirical study to understand code smell distribution on open source Android projects.

Our research contribution provides the community with an open source tool to either to build upon or to use in existing projects. We also perform an empirical study to understand android code smells. Since the existing empirical studies use a small project to perform their study, we aim to conduct our empirical study on a large-scale dataset. This research

provides insight of the smells impact and better understanding of smells to be avoided.

Chapter 2

Related Work

Research into Android related source code best practices has taken many forms, with researchers either studying the occurrence and impact of traditional code smells in Android apps or defining smells that are specific to the Android environment and the development of tools to detect specific smell types. Android code smells are bad implementation practices within Android applications that may lead to poor software quality [3]. The tool is proposed called HOT-PEPPER to automatically correct code smells and evaluate their impact on energy consumption. The tool is limited to refactoring specific code smell Internal Getter/Setter, Member Ignoring Method, and HashMap Usage. Refactoring these code smells effectively and significantly reduces the energy consumption of Applications.

aDoctor, a tool with the ability to detect 15 Android specific code smells was developed by Palomba et al. [16]. With an average precision and recall of 98%, the tool detects smells across multiple categories such as Energy Efficiency, and Energy Efficiency. Colton et al. [5] provided the community with a tool, P-Lint, that, evaluates the source code of an app for occurrences of 16 bad permission-related coding practices (i.e., permission smells). Another static analysis tool, M-Perm, developed by Chester et al. [4] compares permissions listed in the AndroidManifest.xml against the permissions used in the source code for occurrences of permission misuse (i.e., dangerous permissions, under privileges and over privileges) in the app. CheckDroid, developed by Yovine and Winniczuk [20], provides developers with the ability to identify a range of bad practices implemented in an app by reverse engineering the app and performing a taint-based analysis. Peruma [18] performed an empirical study on the occurrence and severity of test smells in open source

Android applications.

In addition to custom-built tools, most IDE's provide in-built support for developers to detect common code quality issues. Integrated into Android Studio is a lint tool[1] to aid developers in the identification and correction of poor coding practices. The tool detects code issues across four categories with the ability for developers to integrate it into the build process. Other popular tools utilized by developers in detecting code quality include (but not limited to) Checkstyle[2], PMD[3], FindBugs[4], and JDeodorant[5]. These tools are utilized primarily for the detection of violations of general Java programming guidelines/standards.

Habchi el al. [7] investigated the presence of smells in iOS apps (Objective-C and Swift). The authors utilized a combination of object-oriented and iOS specific smells in their study. Through a means of comparison, the authors observed that Android apps tend to be more smelly than iOS apps.

A large-scale empirical study on the impact of smells on the quality metrics of Java Mobile Edition based applications was conducted by Linares-Vásquez et al. [13]. Through their work, the authors observed that the occurrence of certain smells is dependent on the domain of the mobile app. However, the authors did not observe correlations between the smells and quality metrics of the analyzed apps.

[1] https://developer.android.com/studio/write/lint/
[2] http://checkstyle.sourceforge.net/
[3] https://pmd.github.io/
[4] http://findbugs.sourceforge.net/
[5] http://jdeodorant.org/

Chapter 3

Android code smells

In this section, we will list all the smells that are detected in Java, XML Layout and AndroidManifest files.We provide a brief description of the 13 Android specific smells, used in this study and also specified in Reimann et al. smell catalog [11]. In addition to describing each smell, we also provide a code snippet containing an example of the smell. The complete smelly code files are available online[1].

3.1 JAVA Smells

3.1.1 Bulk Data Transfer On Slow Network

Occurs when transferring data over a slower network connection. This action will consume much more power than over a fast connection. Reimann et al's catalogue claimed that sending 90kbps over EDGE network's type will consume 300mA * 9.1 min = 44 mAh while sending 300kbps over 3G network's type will consume 210mA * 2.7 min = 9.5 mAh, which is less power compared to sending data over slow internet connection. It is important for a developer to check the device's connectivity before sending data. Fail to check the network type of class used the Internet APIS will be determined smelly.

To detect this smell, the tool performs two steps to detect if the file is smelly or not. First, it searches for a variable that has a type of HttpURLConnection or HttpClient to determine if the class is accessing the internet since these are the APIs that are needed to establish an internet connection. If the class has one of the variables of either of these two

[1] https://drive.google.com/drive/folders/1QPSmhLL8xOWYzLOONLizWE8smTJqE5J5

Table 3.1: JAVA, XML and AndroidManifest smells detection rules

JAVA Smell	**Detection Rule**
Bulk Data Transfer On Slow Network	Determines if a class deal with internet connection then check if inside the class there is not checking Internet connection type.
Prohibited Data Transfer	Determines if class deal with the internet connection and check if the class does not check if the Internet is not disabled.
Interrupting From Background	Determines if the class implement BroadcastReceiver and Service then check if there are start activities on OnRecive method.
Dropped Data	Determines if any user input controls were used in a class were not utilized inside *onSaveInstanceState* method.
Early Resource Binding	Determines if *requestLocationUpdates* method call was used in *OnCreate* method.
Uncached Views	Determines a class has *getView* method and View was set every time the ListView is being scrolled.
Tracking Hardware Id	Determines a class has TelephonyManagerġetDeviceId() or getImei().
Overdrawn Pixel	Determines a class has *onDraw* and does not call *clipRect()* method.
XML smells	
Nested Layout	Determines an element has attribute call *layout_weight*.
Uncontrolled Focus Order	Determines an UI element does not used directional controls.
Untouchable	Determines a UI elements' width be less then 48dp.
Not Descriptive	Determines an UI elements that doen not have *contentDescription*.
AndroidManifest smell	
Set Config Changes	Determines if AndroidManifest has *configChanges* attribute.

datatypes the tool proceeds to perform a second search. This search looks inside each IF condition for NETWORK_TYPE_LTE or NETWORK_TYPE_UMTS and TYPE_WIFI to determine if the class has performed a network type check before sending data. If the tool detects these types, the class will be marked as not smelly. Otherwise, it will be considered as a smelly class. As shown in the given example 3.1, the class is fetching data without checking the internet speed which makes the class smelly.

```
/*
    ** The method contains HttpURLConnection indicate the class dealing with Internet.**
*/
      private InputStream fetch(String source) throws IOException {

         OkHttpClient client = Utils.createOkHttpClient();
         HttpURLConnection connection = client.open(new URL(source));
         return connection.getInputStream();
      }
/*
    ** The whole class does not check the Internet connectivity **
*/
```

Listing 3.1: Example - Bulk Data Transfer On Slow Network.

3.1.2 Prohibited Data Transfer

Occurs when developers do not check internet status before transmitting data in background or foreground. In other words, the developers have to check if the user disables internet connectivity before attempting to send any data in the background or foreground.

To detect this smell, the tool performs two steps to identify if the class is smelly or not. First, it scans if the class requires an internet connection by following the same step done in *Bulk Data Transfer On Slow Network* smell. If the class does require internet access, it will perform the second step by looking for a variable of type **NetworkInfo** and obtains the variables name. After getting the name, the tool scans each if condition to see if the developer performs a null check on the variable. If a null check is performed, the class is marked as not smelly. As shown in the given example 3.1, the class is fetching data without checking the internet is available which makes the class smelly.

3.1.3 Interrupting From Background

Occurs when Activities start from **BroadcastRecievers** or **Services** that work in the background. This smell will result in the user being interrupted from performing a task when a broadcast notification is received. To detect this smell, the tool scans the class for an implementation of either `BroadcastReceiver` or `Service`. If found, the tool searches for

the the method, `onReceive()`, It considers the class smelly if the class has one of the following: startActivities, startActivityFromChild , startActivity and startActivityFromFragment. As shown in example of Interrupting From Background smell, the class is smelly because `onReceive()` method has call to startActivity().

```
public class OnBootReceiver extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent intent) {

        final String action = intent.getAction();

        if(Intent.ACTION_BOOT_COMPLETED.equals(action)) {
            VpnProfile bootProfile = ProfileManager.getOnBootProfile();
            if(bootProfile != null) {
            Log.i(TAG, "starting profile '" + bootProfile.getName() + "' on boot");
            Intent startVpnIntent = new Intent(context, GrantPermissionsActivity.class);
            startVpnIntent.putExtra(context.getPackageName() + GrantPermissionsActivity.EXTRA_UUID,
            bootProfile.getUUIDString());
            startVpnIntent.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
            //startVpnIntent.putExtra(LogWindow.EXTRA_HIDELOG, true);
            context.startActivity(startVpnIntent);
            } else {
            Log.d(TAG, "no boot profile configured");
            }
        }
    }
    }
```

Listing 3.2: Example - Interrupting From Background.

3.1.4 Dropped Data

Occurs when developers do not implement onSaveInstanceState(), which is a Bundle object containing the activity's previously saved state, and ensures that the user data is saved. When the Activity is interrupted the onSaveInstanceState() method is triggered; if the developers implemented the logic to save the input data, it would not be a smell. In the example below, the developer did not override onSaveInstanceState() even though the class deals with user inputs (EditText). The code given below was reduced due to class size; we only keep the method name and removing the body to show that the developer did not take into account saving user input.

To detect this smell, the tool obtains the name of the variables if a class contains the following datatypes: TextInputEditText, EditText, NumberPicker, Button, ToggleButton, CheckBox, RadioButton, Spinner, SeekBar, DatePicker, and TimePicker. The tool will check the body of the onSaveInstanceState() method if the value of these input controls are being used. If not, the class is marked as smelly.

```
public class FormFragment extends BaseFragment implements LoaderManager.LoaderCallbacks<Integer> {

    private TextView mTitle;
    private EditText mEditTitle;
    private EditText mEditText;

    @Override
    public void onAttach(Activity activity) {

    @Override
    public Loader<Integer> onCreateLoader(int i, Bundle args) {

        return 1;
    }
    @Override
    public void onLoadFinished(Loader<Integer> sender, Integer pid) {
    }

    public void clearForm() {
    }

    public void hideKeyboard() {
    }

    @Override
    public void onLoaderReset(Loader<Integer> sender) {
    }
}
```

Listing 3.3: Example - Dropped Data

3.1.5 Early Resource Binding

Occurs when energy-consuming physical components of an Android device are requested too early. This action will result in more energy will being consumed. For example, requesting the GPS component of an Android device in the onCreate() method will result in a waste of energy since there will be nothing visible to the user when this method is executed. In our study, we detect if the GPS has been requested within the onCreate() method.

To detect this smell, the tool scans for call method name requestLocationUpdates inside the onCreate() method. If requestLocationUpdates method was called inside onCreate(), the class is smelly. The GPS should be requested when the view loads for a user. As provided in the example below, we observe that the class is smelly due to a call to requestLocationUpdates inside onCreate ().

```
public class PointingLocationActivity extends Activity{

    //this variabled are to manage the GPS-GPSListener
    private LocationManager mlocManager = null;
    private LocationListener mlocListener = null;

    private Location location = null;
```

```
    /** Called with the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.pointer);
        Bundle extras = getIntent().getExtras();

        palina = PalinaList.getById(palinanr);
        //creating the listener for the GPS
        mlocManager = (LocationManager)getSystemService(Context.LOCATION_SERVICE);
        mlocListener = new MyLocationListener();
        mlocManager.requestLocationUpdates( LocationManager.GPS_PROVIDER, 0, 0, mlocListener);

        sensorService = (SensorManager) getSystemService(Context.SENSOR_SERVICE);
        Sensor sensor = sensorService.getDefaultSensor(Sensor.TYPE_ORIENTATION);
        if (sensor != null) {
            sensorService.registerListener(mySensorEventListener, sensor,
                    SensorManager.SENSOR_DELAY_NORMAL);
            Log.i("Compass MainActivity", "Registerered for ORIENTATION Sensor");

        } else {
            Log.e("Compass MainActivity", "Registerered for ORIENTATION Sensor");
            Toast.makeText(this, "ORIENTATION Sensor not found",
                    Toast.LENGTH_LONG).show();
            finish();
        }
    }


}
```

Listing 3.4: Example - Early Resource Binding

3.1.6 Uncached Views

Occurs when findViewById () is called in getView, which is a method responsible for creating item's view of a ListView, frequently while scrolling a ListView or switching between pages of ViewPager. This could lead to slow behavior if the developer did not check if the view was already created. The developer should reuse the view instead of creating a new one.

To detect this smell, the tool scans the class override methods to find the method called *getView*, which is a method responsible for setting the content of the individual views and accepting a list of parameters. Once the tool finds the method, it obtains the name of View parameter to check if the view name was used in a null condition check. The tool will mark the class smelly if the obtained name was not found in any If conditions that performed a null condition check.

```
    @Override
    public View getView(int position, View rowView, ViewGroup parent) {

            LayoutInflater inflater = (LayoutInflater) context
                    .getSystemService(Context.LAYOUT_INFLATER_SERVICE);
```

```
            rowView = inflater.inflate(R.layout.tide_item, parent, false);

            ViewHolder viewHolder = new ViewHolder();
            viewHolder.headers = new TextView[6];
            viewHolder.datas = new TextView[6];
            viewHolder.dayTV = (TextView) rowView.findViewById(R.id.tideHeader);

             rowView.setTag(viewHolder);
        }
```

Listing 3.5: Example -Uncached Views

3.1.7 Tracking Hardware Id

For some use cases, it might be necessary to get a unique, reliable, unique device identifier. This could be achieved by reading the IMEI, MEID, or ESN of the phone by calling TelephonyManagerġetDeviceId() or getImei() (Note: getDeviceId() was deprecated in API level 26). These methods require the permission READ_PHONE_STATE. Not only is this an invasion of user privacy it is not stable or reliable. The developer should use an alternative way to get the unique device id.

To detect this smell, the tool scans for TelephonyManagerġetDeviceId() or getImei() inside the class. If the class calls one of them to obtain a unique id, the class will consider smelly. As shown in the example, we can observe the class contains a call to getDeviceId() which make the class smelly.

```
public void onCreate() {
      Log.d(LOGTAG, "onCreate()...");
      telephonyManager = (TelephonyManager)

      // Get deviceId
      deviceId = telephonyManager.getDeviceId();
      // Log.d(LOGTAG, "deviceId=" + deviceId);

  }
```

Listing 3.6: Example -Tracking Hardware Id

3.1.8 Overdrawn Pixel

Occurs when developers override a view onDraw(Canvas canvas) without using canvas.clipRect(). By default [6], the Android system deals with overdraws by removing invisible surface thereby wasting process time if the view was not customized. Once a view is customized

by overriding the view using onDraw(Canvas canvas), the system may not be able to remove any overlaps that occur. As shown below the developer did not call canvas.clipRect() to clean the unseen surface.

To detect this smell, the tool scans the class override methods to find method call *onDraw* which is a method responsible for drawing a custom view and accept one parameter as shown in the example. Once the tool find the method, it search for *clipRect()* is being used inside method. The tool will mark the class smelly if the *clipRect()* was not found.

```
@Override
protected void onDraw(Canvas canvas) {
    if (bitmap != null) {
        canvas.drawBitmap(bitmap, 0, 0, null);
    }

    for (Pair<Path, Paint> p : paths) {
        canvas.drawPath((Path) p.first, (Paint) p.second);
    }

    onDrawChangedListener.onDrawChanged();
}
```

Listing 3.7: Example -Overdrawn Pixel

3.2 XML Smells

3.2.1 Nested Layout

Occurs when layouts with elements that have the attribute **weight** set must be computed twice. Since each new element requires initialization, layout, and drawing, parsing deep nested LinearLayouts will increase the computation time exponentially.

To detect this smell, the tool scans inside each layout element for attribute call *layout_weight*. The tool considers the file smelly if one of the element has it. As shown below in the example, LinearLayout contains the element of EditText type has layout_weight which make this file smelly.

```
<LinearLayout android:layout_height="wrap_content"
              android:orientation="horizontal"
              android:layout_width="fill_parent">
<EditText android:layout_height="wrap_content"
          android:layout_weight="1"
          android:layout_margin="5dip"
          android:imeOptions="normal"
          android:inputType="text">
</EditText>
</LinearLayout>
  <ListView
```

```
        android:id="@+id/m_lv_packages"
        android:layout_height="0dp"
        android:layout_weight="1" >
    </ListView>
</LinearLayout>
```

Listing 3.8: Example -Nested Layout

3.2.2 Uncontrolled Focus Order

Occurs when a developer is not using directional controls for UI elements. There are four directions of navigation up, down, left (previous) and right (next). By default, the Android system computes the nearest neighbor UI element and sets it appropriately. In some cases, the Android system may not be able to make sense of the navigation direction. Therefore, each UI elements should have one direction. An example is provided in Listing 3.8, all the UI elements do not have one of four directional controls. In this case, the XML file has two smells nested layout and Uncontrolled Focus Order.

To detect this smell, the tool scans inside each UI element for one of the following:*nextFocusUp*,*nextFocusDown*,*nextFocusLeft*,*nextFocusRight* and *nextFocusForward* attribute value. The tool consider the file smelly if one of them was not used. As shown below in the example, LinearLayout contain element of EditText type has layout_weight which make this file smelly.

3.2.3 Untouchable

When developers set up UI elements' width to be less then 48dp (ca. 9mm). An example is provided in Listing 3.9, we can observe ColorButton width is less then 48dp which make the file smelly.

To detect this smell, the tool scans inside each layout element for attribute call *layout_width* and gets the value of it. The tool checks if the value is less than 48dp and considers the file smelly if it is. As shown in the example below, ColorButton has a width of 40dp which is less 48dp which make this file smelly.

```
        <android.support.v7.widget.CardView
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:layout_marginTop="16dp"
```

```
        card_view:cardElevation="1dp"
        card_view:cardCornerRadius="4dp"
        card_view:contentPadding="10dp"
        android:orientation="horizontal">
        <LinearLayout
            android:orientation="horizontal"
            android:layout_width="match_parent"
            android:layout_height="wrap_content">
            <at.bitfire.icsdroid.ui.ColorButton
                android:layout_width="40dp"
                android:layout_height="40dp"
                android:id="@+id/color"
                android:layout_gravity="center"
                android:layout_marginLeft="16dp"/>
        </LinearLayout>
    </android.support.v7.widget.CardView>
```

Listing 3.9: Example -Untouchable

3.2.4 Not Descriptive

Occurs when developers do not add a description for UI elements in the layout XML file to add meanings for the UI. The UI description will be read by TalkBack which covert the text written in the description to voice to help people who have a vision disability understand the purpose of the UI.

To detect this smell, the tool scans inside each UI element for attribute call *contentDescription*. The tool consider the file smelly if one element does not have *contentDescription*. An example is provided in Listing 3.8 that shows the lack of adding *contentDescription* to UI elemnts(EditText) which make the file smelly.

3.3 AndroidManifest Smell

3.3.1 Set Config Changes

Occurs when developers are trying to handle some configuration manually instead of allowing the operating system to control the behavior. This action may result in memory bugs due to developers not releasing the resources in memory. This can be accomplished by adding the configChanges attribute to the AndroidManifest file. As shown in the example provided in Listing 3.10, the developer wants to handle keyboardHidden, orientation and screen size manually.

To detect this smell, the tool scans inside *AndroidManifest xml file* for attribute call *configChanges*. The tool consider the file smelly if has this attribute *configChanges*.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.dev.cromer.jason.cshelper"
    android:configChanges="keyboardHidden|orientation|screenSize"
    android:screenOrientation="portrait"
    android:windowSoftInputMode="stateHidden">
```

Listing 3.10: Example -Set Config Changes

3.4 Android Code Smells Detector

We introduce a new Android code smells detector, coined *BadDroidDetector*, that can identify 13 Android-specific code smells provided by Reimann et al. [11]. The tools is an open-source that can run quickly by giving the list of file that you need to analyze. After providing the list of files to be scanned, *BadDroidDetector* will automatically scan all given input and populate the same list along with the occurrences of all smell types. Internally, *BadDroidDetector* utilizes JavaParser [2] to parse the Java source file through the use of an abstract syntax tree (AST). Depending on the type of smell being detected, we override the appropriate `visit()` method to perform our analysis/detection. The design of *BadDroidDetector* is such that it facilitates the inclusion of detection rules for additional smell types. The output from *BadDroidDetector* is in the form of a CSV file. Each row in the CSV file corresponds to a unit test file, and the associated columns contain boolean values indicating if the specific smell type is present or not.

[2] `https://javaparser.org/`

Chapter 4

Methodology

In our study, we want to investigate the occurrence of Android code smells and their impact on the overall quality of Android applications. To achieve this, we conducted a two-phased approach that consisted of (1) data mining and (2) smells detection.

4.1 Research Questions

We investigate the occurrence of Android specific code smells in Android apps and their impact on the overall quality of the apps through a set of quantitative, comparative and empirical experiments that answer the following research questions:

- **RQ1:** ***What is the frequency and distribution of Android specific smells in apps?*** We intend to better understand the existence and frequency of smells occurrences by analyzing a larger dataset of open-source mobile projects. The purpose of this analysis is to verify whether the bad programming practices, mentioned in the catalog are actually practical in the sense of them describing real-world programming scenarios.

- **RQ2:** ***To what extent does the severity of smells increase the risk of files change- and bug-proneness?*** This research questions investigates the impact of these bad programming practices to verify whether they have a negative impact on the project maintainability. To do so, we analyze cluster the project files into infected (with smells) and noninfected, then we verify whether infected files tend to be more expensive in terms of maintenance. This is achieved by empirically checking whether

they are more prone to code changes and to bugs in comparison with the noninfected files.

4.2 Data Mining Phase

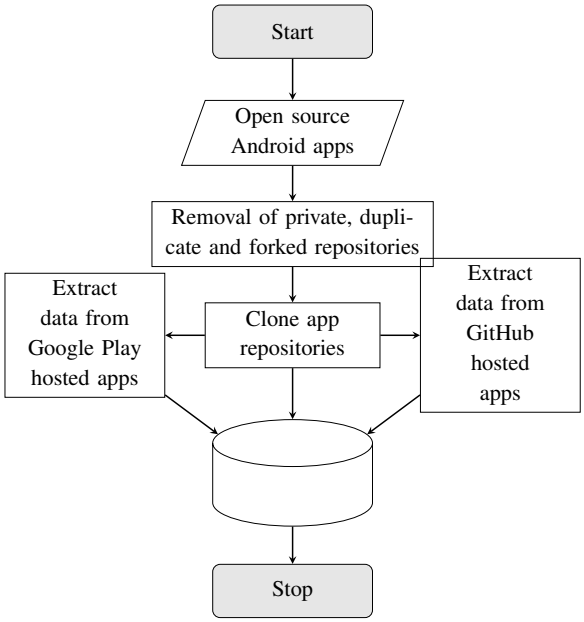

Figure 4.1: Overview of the Data Mining Phase

Table 4.1: Overview of data obtained in the Data Mining Phase

Item	**Value**
Total cloned repositories	2,011
Cloned apps hosted on GitHub	1,835
Cloned apps utilizing GitHub's issue tracker	808
Total number of commit log entries	1,037,236
Total number of Java files affected by commits	6,379,006
Total volume of repositories cloned	53.8 GB

For data mining, we were able to obtain a copy of the same dataset that used in [18, 17].

During the data mining process, [18] used F-Droid [1] to obtain the index of open-source Android apps to clone what is available on Git-based version control systems excluding duplicated and forked. For each of the cloned repositories, [18] was able to result in the following data : (1) the entire commit log (2) list of all files affected by each commit, (3) all available tags as for projects hosted on GitHub, the author, the popularity metrics (including the number of Stargazers, Forks, Subscribers, and Releases) and issue tracker details associated with each project. Depicted in Figure 4.1 is an overview of the Data Mining Phase process, while Table 4.1 provides an overview of the data collected.

4.3 Detection Phase

The purpose of this phase is to detect Android smells that take place in either JAVA, XML OR AndroidManifest files. Our files type selection is based on our analyzing of the definition of the Quality Smells Catalogue [11] addressed by Reimann et al.

The detection phase carries out multiple steps to detect the smells. Firstly, we detect and separate all the JAVA, XML and AndroidManifest files using different parsers. Secondly, the result of detecting and separating the files has been saved in different comma separated values files (CSVs). Finally, we fed the CSV files to *BadDroidDetector* to identify the smells in each file's types. To do that, a helper tool was built to assist identifying and separating all the files that in the form of JAVA, XML and AndroidManifest type. As depicted in Figure 5.2, we first identify JAVA and XML which exist throughout the lifetime of the app. Next, we check if the files are parsable then we save the list of identified JAVA and XML files into CSV file and use them as input for *BadDroidDetector*. More detail will be provided in each of the detection activities. An overview of the data collected/analyzed in this phase is provided in Table 4.2.

[1] `https://f-droid.org/`

Figure 4.2: Overview of the Detection Phase

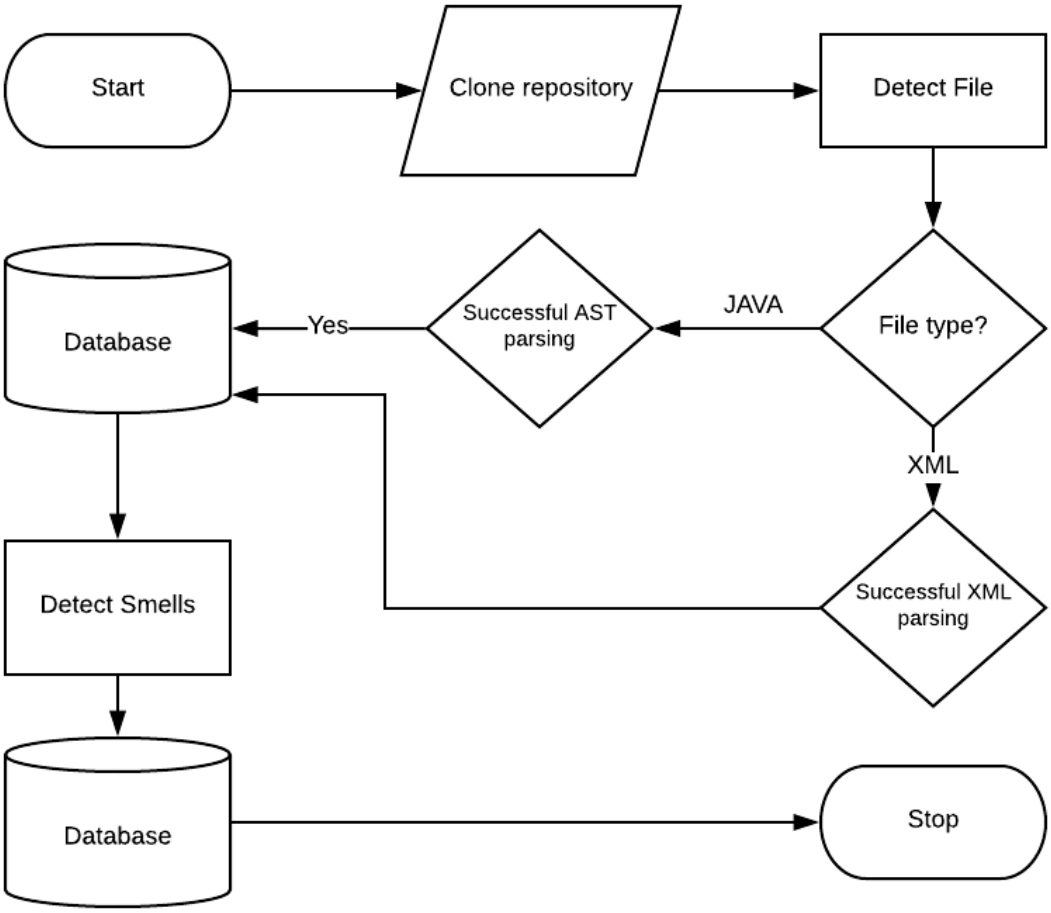

Figure 4.3: Overview of the Detection Phase

4.3.1 Files Detection

As we illustrate at the beginning of the section, before detecting the code smells, we first identified and saved the java and XML files. The file detector tool is a standalone tool that only takes the directory's path that has all application needed to be scanned. Below a description of how we detected JAVA and XML files.

Table 4.2: Overview of cumulative data obtained in the Detection Phase throughout all versions of projects

Item	Cumulative	Latest Version
Java		
Total Files	1931175	268814
Files not exhibiting any smells	1611286	24431
Files containing 1 or more smells	306743	24503
Files containing 2 or more types of smells	27382	2081
Files containing 3 or more types of smells	1157	66
Files containing 4 or more types of smells	0.00	0.00
Files associated with a GitHub issue	94120	7852
XML		
Total Files	366527	61782
Files not exhibiting any smells	0	0
Files containing 1 or more smells	366527	61782
Files containing 2 or more types of smells	365769	61554
Files containing 3 or more types of smells	118159	14283
Files associated with a GitHub issue	12652	1671
AndroidManifest		
Total Files	98126	5871
Files not exhibiting any smells	98095	5871
Files containing 1 or more smells	31	2
Files associated with a GitHub issue	2994	114

4.3.1.1 JAVA File

After directory's path as assigned as input to the tool, the tool will start scanning inside each folder and capture the file the has .java extension. Once the tool has found a java file, the tool parses the java file by using JavaParser to obtain an AST to make sure the file syntax is correct. If the file was parsed correctly, we do further analysis to obtain app name where the file was found, relative file path and full path, file name, associated XML file, total imports, TotalMethodStatements, and PackageClassName. After obtaining all these

data; the result will be appended into CSV file that will have all scanned java files that exist in the given directory.

4.3.2 XML File Detection

The same tool was used to scan JAVA files, utilize to scan XML file as well. We used a different parser to parse an XML file. We choose dom4j [**?**] to parse XML files since it is the most popular framework for Java. Before parsing XML file, the tool checks if the file name is *AndroidManifest* or the file's location within the following locations: res/layout/ or resources/layout/ to start parsing the XML file by utilizing Dom4j for parsing. It is essential only to scan the XML files that deal with UI in activity or fragment components. As android documentation addressed [1], the architecture layout resources for the UI are in res/layout/ or resources/layout/. After the parsing done, the tool will extract the following date: app name where the file was found, relative file path and full path, file name and save the obtained data by following the same mechanisms we did JAVA file detection.

After the file populated, we import the data from CSV file to the MySQL database to extract all the *AndroidManifest* files into separate CSV file so we can use it as input for *BadDroidDetector*. Once we have the list of all AndroidManifest, we create another CSV file that has all scanned file without AndroidManifest. By doing so, we will have two CSV files that can be used as an input for *BadDroidDetector*. More details will be provided in Android Code Smells sections.

4.3.3 Android Code Smell Detection

After detecting JAVA and XML files, we ran the *BadDroidDetector* three time to identify the occurrence and distribution of Android Code Smells. One ran for the JAVA file smells and others for AndroidManifest and XML smells.

BadDroidDetector utilized JavaParser when the ran was for JAVA smells depending on type smell, we override the appropriate `visit()` method to perform our analysis and detection. Results provided by *BadDroidDetector* were saved in a CSV file for analysis

and interpretation.

Utilizing Dom4j, *BadDroidDetector* parses the XML source file. Depending on the type of smell being detected, we override the `runAnalysis()` method to perform our analysis/detection rules. Results provided by *BadDroidDetector* were saved in a database for analysis and interpretation.

Chapter 5

Analysis & Discussion

In this chapter, we present answers to our research questions by analyzing the occurrence and impact of test smells in the studied apps.

5.1 RQ1: What is the frequency and distribution of Android specific smells in apps?

Motivation: Most of the prior research on the code quality of Android apps has focused on the existence of traditional code smells in the source code of the app [14]. Similar to this, we aim to understand the degree to which Android specific code smells are present in the apps source code. The results of this RQ aims to better inform developers on the most common type of Android code smells that are most likely to be present in their codebase and be in a better position to address these issues earlier on in the development lifecycle. To this extent, we investigate the distribution and the frequency of occurrence of Android-specific smells in the apps in our study. This RQ also involves the study of co-occurrence of smells as a means of investigating the relationship among the various smells.

Approach: Our approach first involved executing our smell detection tool on our corpus of apps. We next performed standard statistical analysis to obtain the distribution of smells as well as the degree to which each smell type occurs in apps and files. Additionally, we also studied the co-occurrence of each smell type. Since the smells in our study impact Java source code files, layout (XML) files and the AndroidManifest.xml file, our analysis and reporting are also categorized into these three categories.

Table 5.1: Distribution of overall smells throughout the lifetime Android apps

Smell Type	**Distribution**
JAVA	
Dropped Data	66.66%
Uncached Views	13.25%
Bulk Data Transfer On Slow Network	7.71%
Overdrawn Pixel	5.73%
Prohibited Data Transfer	3.29%
Interrupting From Background	1.73%
Tracking Hardware Id	1.53%
Early Resource Binding	0.09%
XML	
Not Descriptive UI	41.72%
Uncontrolled Focus Order	41.81%
Untouchable	8.67%
NestedLayout	7.79%
AndroidManifest	
Set Config Changes	N/A

Result:

Presented in Table 5.1, are the distribution of smells in our studied apps. We also provide details of the occurrence of each smell type in apps and files in Table 5.2 as well as smell co-occurrence in Tables 5.3 and 5.4.

For the smells that are detected in Java files, we observed that the most occurring Android smell is *DroppedData*, when compared with the other smells. Furthermore, the *DroppedData* smell occurred in over 79% of the overall analyzed apps. The reason behind this phenomenon is due to developers assuming that users will continually be using the app (and not accounted for interruptions). Interestingly, we observed that the smell *UncachedViews* has a high co-occurrence with this smell. The reason for this could be due to both smells being associated with some form of user input/interaction. Since the

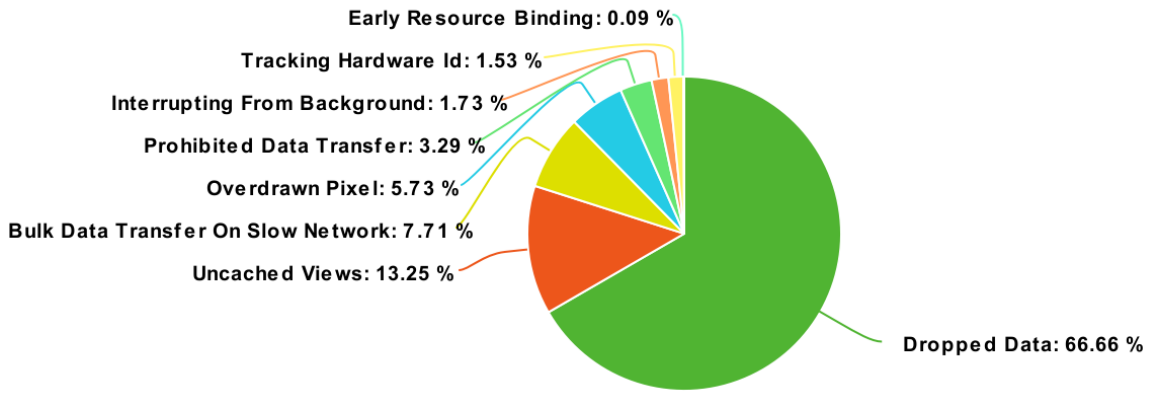

Figure 5.1: Distribution of overall Java Layout smells throughout the lifetime Android apps

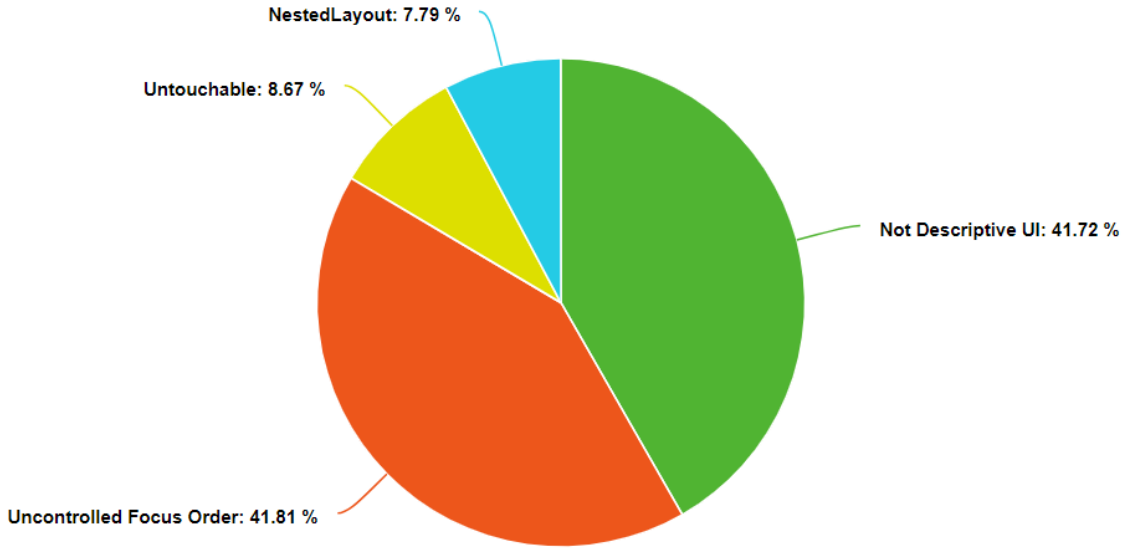

Figure 5.2: Distribution of overall XML Layout smells throughout the lifetime Android apps

smell *UncachedViews* is related to the UI (particularly to ListViews), the high occurrence (13%) of it is also not surprising, since most of the apps have some form of user interaction. This smell indicates that developers force the ListView to create items of a view when users scroll or switch between pages. Interestingly, even though the smell *Overdrawn Pixel* appears in approximately 29.65% of the apps, the spread of the smell in the source Java

Table 5.2: Occurrence per App and File

Smell Type	Smell Occurrence Per App	Smell Occurrence Per File
JAVA		
Dropped Data	79.32%	11.65%
Uncached Views	34.64%	2.32%
Overdrawn Pixel	29.65%	1.00%
Bulk Data Transfer On Slow Network	28.08%	1.35%
Prohibited Data Transfer	21.97%	0.58%
Interrupting From Background	9.42%	0.30%
Tracking Hardware Id	4.76%	0.27%
Early Resource Binding	1.35%	0.02%
XML		
Not Descriptive UI	100%	99.78%
Uncontrolled Focus Order	100%	99.99%
NestedLayout	61.32%	18.62%
Untouchable	51.56%	20.74%
AndroidManifest		
Set Config Changes	0.10%	0.031%

files is low. This could be due to developers opting to manually create the UI instead of a dynamic (code-based) approach. Most likely the UI's are simple or developers prefer a no-code based approach.

We observed an extremely high co-occurrence, of over 90%, between *Prohibited Data Transfer* and *Bulk Data Transfer On Slow Network*. This high co-occurrence is not surprising as both smells are related to networking. This indicates that developers ignore checking the internet status before transmitting data in the background, and also ignore selecting the network connection type (e.g. EDGE, 3G and WiFi) at the same type.

When it comes to XML layout based smells, we found that *Not Descriptive UI* and *Uncontrolled Focus Order* are the two smells that occur the most. These two smells are accessibility related and will negatively impact disabled individuals trying to use the app.

Table 5.3: Co-occurrence of Java smells

Smell Type	**BDTSNR**	**DD**	**ERB**	**IF**	**OP**	**PDT**	**THI**	**UV**
BulkDataTransferOnSlowNetwork		10.90%	0.02%	0.21%	0.08%	42.01%	0.53%	1.38%
DroppedData	1.26%		0.00%	0.06%	0.41%	0.41%	0.35%	5.57%
EarlyResourceBinding	1.37%	2.73%		0.00%		0.34%	0.00%	0.00%
InterruptingFromBackground	0.95%	2.30%	0.00%		0.00%	0.24%	0.43%	0.00%
OverdrawnPixel	0.10%	4.80%	0.00%	0.00%		0.00%	0.25%	0.64%
ProhibitedDataTransfer	98.32%	8.23%	0.01%	0.13%	0.00%		0.05%	0.04%
TrackingHardwareId	2.68%	15.07%	0.00%	0.49%	0.95%	0.10%		0.37%
UncachedViews	0.80%	28.04%	0.00%	0.00%	0.28%	0.01%	0.04%	

Abbreviations:
BDTSNR = BulkDataTransferOnSlowNetwork
DD = DroppedData
ERB = EarlyResourceBinding
IF = InterruptingFromBackground
OP = OverdrawnPixel
PDT = ProhibitedDataTransfer
THI = TrackingHardwareId
UV = UncachedViews

Table 5.4: Co-occurrence of Layout smells

Smell Type	**NestedLayout**	**UncontrolledFocusOrder**	**NotDescriptiveUI**	**Untouchable**
NestedLayout		99.99%	99.79%	38.24%
UncontrolledFocusOrder	18.62%		99.78%	20.74%
NotDescriptiveUI	18.65%	99.99%		20.77%
Untouchable	34.34%	99.98%	99.93%	

Developers should strive to ensure that the apps being designed and built adhere to accessibility standards and guidelines, and thus be usable by all users, regardless of disabilities [2]. However, our study highlighted that his not the case in reality. Out analysis of over 1,975 apps indicated that at least one file, in each app, contained either both or one of these accessibility smells. Looking at the co-occurrence of these two smells we noticed that **90%** each smell co-occurs with the other. This further provides us with more evidence that developers do not pay attention to accessibility considerations in their apps. It is also interesting to note that the other layout smells, also have a high co-occurrence with the accessibility smells. This further highlights the lack of accessibility support in UI's.

For our AndroidManifest smell, we obtained an interesting finding when we only detected 31 instances of **Set Config Changes** which only appears in two apps from 1,975 apps. The **Set Config Changes** smell occurs when a developer requests the Android operating system to stop automatically handling some configuration changes (like orientation change, font size change) and instead the developer manually makes these changes. This action will introduce memory bugs by not clearing the resources in memory. A small occurrence of this smell indicates that developers prefer to allow the operating system to handle configuration changes rather than implementing custom code. However, we will conduct further studies on these two apps on GitHub's issues to know if the app has issues related to memory when the first time the smell was introduced. Also interesting is that both of the smells occur once the file is created.

5.2 RQ2: To what extent does the severity of smells increase the risk of files change- and bug-proneness?

Motivation: Through RQ1 we observed a widespread occurrence of Android smells in our large corpus of apps. However, to further highlight that these smells impact both code quality and maintainability we investigate the degree to which these smells increase the likelihood of change- and bug-proneness of files.

Approach: To answer **RQ2**, we compute the Odds Ratio (OR) for all the smells, similar

to the prior work [12]. As stated in Chapter 5, we obtained the dataset of the commits and issues for each app that we cloned. To calculate the OR for the change-proneness, we need to prepare the data before computing the OR. We count the distinct commits for each distinct file over the lifetime of the app in order to count how many time the file was changed. By obtaining this data, we have a list that contains the file and the number of changes that happen for each file along with a total number of smells for each smell type. From this resultset, we execute queries to get for each smell type the number of the files that have the following attributes (1) has a smell and has > 1 commits, (2) has a smell and no commits, (3) has no smell no commits, and (4) has no smell and has > 1 commits.To measure the OR for bug-proneness, we followed a similar approach. We retrieved the issue tracker details of apps from their GitHub repository. Through queries, we obtained a count of files: (1) has a smell and has issues, (2) has a smell and no issues, (3) has no smell and no issues, and (4) has no smell and no issues.

Result: Table 5.5 shows the result of computing OR for change/ bug proneness for each smell type. The table contain list of smell names along with the OR change/ bug proneness.

Results from our change-proneness study showed that files infected with almost all smell types were prone to changes. Files infected with the smell *Interrupting From Background* were more prone to changes. This is interesting since this smell is not as widespread in our corpus when compared to the other smells. A possible reason for this is due to developers embedding business logic into this file and hence this file undergoing multiple revisions. We noticed that the OR values for the smells associated with networking were more-or-less the same. This indicates that these smells are related, and once again shows that during the development of the app developers will be frequently updating files that contain these smells.Not surprisingly, the XML files show high change-proneness due to the smells being UI related. During the development of the app, the developer would be making multiple updates to the UI based on either requirements or user feedback changes. These files also demonstrated, relatively, high OR values for bug-proneness. This is reflective of the nature of mobile apps. Mobile apps tend to be UI heavy and therefore the

Table 5.5: Odds Ratio for bug and change-proneness. Bold indicates that the obtained result was statistically significant ($p < 0.05$).

Smell Type	Odds Ratio	
	Change Proneness	Bug Proneness
JAVA		
Interrupting From Background	**3.20**	0.04
Early Resource Binding	**2.74**	0.15
Dropped Data	**2.64**	0.80
Bulk Data Transfer On Slow Network	**2.07**	0.60
Prohibited Data Transfer	**2.26**	0.54
UncachedViews	**1.81**	0.52
OverdrawnPixel	1.03	0.53
TrackingHardwareId	0.000048	**1.14**
XML		
Uncontrolled Focus Order	4.05	2.10
Not Descriptive UI	**2.65**	**1.76**
Untouchable	**2.25**	**1.45**
NestedLayout	**1.53**	0.83
AndroidManifest		
Set Config Changes	2.97	2.23

issues raised by the users are related to the UI that the user interacts with. Though the AndroidManifest smell shows a high OR value for change- and bug-proneness, it is not statistically significant. This is due to the low number of apps/files exhibiting this smell. Further research in this area is required. More research is also required on the smell *TrackingHardwareId*, as it shows a statistically significant bug-proneness OR value but an extremely low change-proneness OR value. Most likely the developer introduced this smell earlier on in the project and due to privacy issues (possibly reported by users), updates the file containing this smell.

Chapter 6

Threats to Validity

In this chapter, we present factors that may impact the applicability of our observations in real-life situations. An argument can be made that our corpus of apps is not entirely representative. However, due to cost constraints, it had to be limited to open source apps. Furthermore, the apps are diverse and have also been used in prior studies. The detection methodology in our tool may also be subject to threats. To counter this, we performed manual verification by the authors and external subject matter experts on the accuracy of tools detection abilities. Further, as an open source tool, we welcome the community to contribute to updates to the tool. Additionally, due to different coding styles by developers, there will be instances where the tool fails to detect smells or detects false positives. Catering to all styles is not feasible and hence this study utilized the rules published in the smells catalog. Not all developers utilize GitHub's issue tracking system, nor do they associate commits with issues. This is a possible reason for the OR values in our bug-proneness study. We would need to extend this initial work by focusing on projects that adhere to well-established software engineering principles/processes.

Chapter 7

Conclusion & Future Work

The goal of this work to extend the aDoctor tool [16] by adding the missing smells that aDoctor was not detecting, and to conduct an empirical study on the 13 smells that we are extending to show the distribution and severity of these smell in Android apps. We introduce a tool that detects 13 smells that were listed in [11] catalog. We ran the tool against a large dataset that contains over 1500 Android apps were listed on F-droid website. we observed that the most occurring An-droid smell is DroppedData, when compared with the JAVA smells while we found that Not Descriptive UI and Uncontrolled Focus Order are the two smells that occur the most layout XML smells.

For future work, we plan to study the whole catalog of 30 smells to see what an impact will cause on Android devices resources. The resources that we will look at are the CPU, GPU, memory and battery usage. Also, we plan to build a plugin for this tool that can scan the project at run-time and give refactoring recommendations to remove smells. By doing this will reduce the number of smells and educate developers who do not know about Android smells.

Chapter 8

Acknowledgement

We would also like to thank all the participants who volunteered in this project for their assistance and advice.

Bibliography

[1] Layout resource ălă Android Developers.

[2] Accessibility in apps: the necessity often forgotten. https://www.netguru.co/blog/accessibility-web-mobile-apps, April 2015. (Accessed on 11/11/2018).

[3] Antonin Carette, Mehdi Adel Ait Younes, Geoffrey Hecht, Naouel Moha, and Romain Rouvoy. Investigating the energy impact of Android smells. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 115–126. IEEE, 2 2017.

[4] Piper Chester, Chris Jones, Mohamed Wiem Mkaouer, and Daniel E. Krutz. M-Perm: A Lightweight Detector for Android Permission Gaps. In *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 217–218. IEEE, 5 2017.

[5] Colton Dennis, Daniel E. Krutz, and Mohamed Wiem Mkaouer. P-Lint: A Permission Smell Detector for Android Applications. In *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 219–220. IEEE, 5 2017.

[6] Google Developers. (13) Android Performance Patterns: Overdraw, Cliprect, QuickReject - YouTube.

[7] Sarra Habchi, Geoffrey Hecht, Romain Rouvoy, and Naouel Moha. Code Smells in iOS Apps: How Do They Compare to Android? In *2017 IEEE/ACM 4th International*

Conference on Mobile Software Engineering and Systems (MOBILESoft), pages 110–121. IEEE, 5 2017.

[8] Hecht and Geoffrey. An approach to detect Android antipatterns, 2015.

[9] Geoffrey Hecht, Naouel Moha, and Romain Rouvoy. An empirical study of the performance impacts of Android code smells. In *Proceedings of the International Workshop on Mobile Software Engineering and Systems - MOBILESoft '16*, pages 59–69, New York, New York, USA, 2016. ACM Press.

[10] Geoffrey Hecht, Romain Rouvoy, Naouel Moha, and Laurence Duchien. Detecting Antipatterns in Android Apps. In *2015 2nd ACM International Conference on Mobile Software Engineering and Systems*, pages 148–149. IEEE, 5 2015.

[11] Jan Reimann, Martin Brylski, and Uwe Aßmann. A Tool-Supported Quality Smell Catalogue For Android Developers - Semantic Scholar. *Softwaretechnik-Trends*, 34, 2014.

[12] Foutse Khomh, Massimiliano Di Penta, and Yann-Gael Gueheneuc. An exploratory study of the impact of code smells on software change-proneness. In *Reverse Engineering, 2009. WCRE'09. 16th Working Conference on*, pages 75–84. IEEE, 2009.

[13] Mario Linares-Vásquez, Sam Klock, Collin McMillan, Aminata Sabané, Denys Poshyvanyk, and Yann-Gaël Guéhéneuc. Domain matters: bringing further evidence of the relationships among anti-patterns, application domains, and quality-related metrics in Java mobile apps. In *Proceedings of the 22nd International Conference on Program Comprehension - ICPC 2014*, pages 232–243, New York, New York, USA, 2014. ACM Press.

[14] U. A. Mannan, I. Ahmed, R. A. M. Almurshed, D. Dig, and C. Jensen. Understanding code smells in android applications. In *2016 IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 225–236, May 2016.

[15] Umme Ayda Mannan, Iftekhar Ahmed, Rana Abdullah M. Almurshed, Danny Dig, and Carlos Jensen. Understanding code smells in Android applications. In *Proceedings of the International Workshop on Mobile Software Engineering and Systems - MOBILESoft '16*, pages 225–234, New York, New York, USA, 2016. ACM Press.

[16] Fabio Palomba, Dario Di Nucci, Annibale Panichella, Andy Zaidman, and Andrea De Lucia. Lightweight detection of Android-specific code smells: The aDoctor project. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 487–491. IEEE, 2 2017.

[17] Anthony Peruma. What the smell? an empirical investigation on the distribution and severity of test smells in open source android applications. In *RIT theses*, May 2018.

[18] Anthony Shehan and Ayam Peruma. What the Smell? An Empirical Investigation on the Distribution and Severity of Test Smells in Open Source Android Applications. Technical report, 2018.

[19] Arnaoudova Venera, Massimiliano Di Penta, and and Antoniol Giuliano. Linguistic antipatterns: what they are and how developers perceive them. *Empirical Software Engineering*, 21(1):104–158, 2 2016.

[20] Sergio Yovine and Gonzalo Winniczuk. CheckDroid: A Tool for Automated Detection of Bad Practices in Android Applications Using Taint Analysis. In *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 175–176. IEEE, 5 2017.