Detecting Code Smells in Android Applications

Master's Thesis

Dustin Lim

Detecting Code Smells in Android Applications

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Dustin Lim
born in The Hague, the Netherlands

TUDelft

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
`www.ewi.tudelft.nl`

Detecting Code Smells in Android Applications

Author: Dustin Lim
Student id: 1514202
Email: lim.dustin@gmail.com

Abstract

Code smells are patterns in programming code which indicate potential issues with software quality. Previous research resulted in the development of code smell detectors: automated tools which traverse through large quantities of code and return smell detections to software developers.

The Android platform has become very popular over the years but is relatively new in terms of scientific research. This research investigates the detection performance of existing detectors, tools that were not designed with Android in mind, on a dataset of Android apps. In addition, multiple smell detectors and source code metrics are combined in a machine learning approach to develop an Android-specific smell detector.

Preface

The final hurdle is often challenging; this is at least true in my experience. Friendly people gave me advice that this hurdle would have its ups and downs. That turned out to be true, but then again I knew that the end goal was certainly worth it. I am very happy to present this master thesis as part of my Master of Science degree in Computer Science at the Delft University of Technology.

One year ago I was in search of a thesis project. I would like to thank Andy Zaidman who helped me in my search and agreed to be my university supervisor. He also helped me connect with Fabio Palomba; his experience related to my thesis subject proved enormously helpful to me. I would like to thank Fabio for agreeing to be my daily supervisor and for having all those meetings with me.

I would like to thank Arie van Deursen and David Tax for joining my thesis committee and taking their time to read what is hopefully pleasant reading material. Finally, I would like to give special thanks to my friends Isabeau Dieleman and Daan van Campen. Isabeau generously offered to read and provide feedback for multiple drafts of my thesis. Something I am very grateful for. Daan was a consistent weekly source of moral support and fun humor.

Dustin Lim
Delft, the Netherlands
January 17, 2018

Contents

Chapter 1

Introduction

This chapter describes the context behind this research, research motivation and an outline of the thesis.

1.1 Research Context

Code smells are patterns in programming code that indicate potential issues with software quality. Code smells are different from software bugs, as they do not make programs run incorrectly. Instead, the quality issues implied by code smells produce software that is harder to develop and maintain. The two concepts are however related, as code smells can make software more susceptible to future software bugs.

Previous research produced several techniques for detecting the various types of code smells [24, 17, 20, 18]. This research was generally done in the context of desktop applications. The goal was to use detection techniques to develop *code smell detectors*: automated tools that traverse through (large quantities of) code and display smell detections. Developers can then decide whether to remove these smells. In contrast to desktop applications, less research has been done on code smell detection for mobile applications.

This thesis explores this mobile domain, in particular for the Android platform. The goal was to perform two initial studies on: (1) the performance of 'traditional' smell detectors on Android apps and (2) the development and testing of an Android-specific smell detector.

1.2 Research Motivation

It is tedious for developers to manually traverse code and find code smells for any decent-sized application. In addition, developers may not be aware of the various code smells and the reasoning behind them. Smell detectors automate this process and can potentially help improve software quality on a large scale. Given the trending popularity of mobile applications, it was therefore interesting to investigate code smell detection for such applications.

1.3 Thesis Outline

The thesis consists of four main chapters. In Chapter 2, background information and related work is presented on code smells and detection techniques. This includes scoping the code smells investigated to five specific types, out of a larger catalogue.

Chapter 3 describes the construction of a dataset of code smells present in a selection of Android apps. This 'truth data' is an important prerequisite to determine the detection performance of tools/techniques with respect to these apps. This enables the two subsequent studies in this thesis.

Chapter 4 covers Study I, on the performance of 'traditional' code smell detectors on Android apps. Such detectors were generally developed to accept desktop software. Note that effective smell detection techniques for desktop software do not necessarily extend to mobile apps.

Chapter 5 covers Study II, on the development and empirical testing of a novel Android-specific smell detector. This study attempts to improve on the detection performance obtained in Study I. The novel detector was constructed using a machine learning approach.

Following the four main chapters, Chapter 6 discusses possible threats to validity. Chapter 7 concludes on findings made and proposes future work.

Chapter 2

Background and Related Work

This chapter provides the background of this thesis. This includes definitions and summaries of all significant concepts. In addition, this chapter also provides references to previous related work.

2.1 Code Smells

The concept of code smells was first introduced by Fowler and Beck (1999) [13]. They defined smells as 'structures in the code that suggest (sometimes they scream for) the possibility of *refactoring*'. Refactoring is the process of restructuring code without changing its outward behaviour. These code structures imply poor software design and can appear 'smelly' to developers inspecting the code. Through refactoring, code smells are removed and as a result software design is improved.

Fowler and Beck presented 22 types of code smells. Each smell contained a description of relevant code structure(s), the design issues implied by the smell and general strategies for refactoring. For the scope of this thesis, a comprehensive investigation of all 22 smells was infeasible. Instead a limited set of five smells was selected, listed in Table 2.1.

Table 2.1: Code smells within scope of thesis.

Code smell	**Overview**
Feature Envy	A method that is more interested in the features of another class than its own parent class.
Large Class	A class that is too bloated in size.
Long Method	A method that is too bloated in size.
Message Chain	A chain of method invocation expressions.
Spaghetti Code	A class with very little structure.

Although each of the 22 code smells imply poor design, their real-world impact on software maintainability is not necessarily the same or significant. The five code smells that were selected have been observed by Palomba et al. to significantly affect software maintainability [19]. Removing Feature Envy reduced the *fault-proneness* of code: the number of bug fixes over time on affected software components. Removing Large Class, Long Method, Message Chain and Spaghetti Code reduced the *change-proneness* of code: the number of code changes (commits) on affected software components. All other things being equal, lower fault- and change-proneness translate to simpler software maintainability.

In summary, the scope of this thesis was limited to the five smells in Table 2.1, which were deemed impactful on software maintainability.

2.2 Definitions of Investigated Code Smells

This section defines the five code smells investigated. Most of the code smell descriptions given by Fowler and Beck [13] contain abstract elements. As a result, they are open to personal interpretation. Smell detection techniques are generally developed by deriving suitable concrete metrics from these abstract descriptions. This section contains the interpretation of the five code smells as used within the thesis. Unless stated otherwise, they were directly summarized from Fowler and Beck.

2.2.1 Feature Envy

Affects class methods. In Object Oriented Programming (OOP), data and related functionality is organized into classes. The Feature Envy smell is present whenever a method located in one class is clearly more interested in the features of another class. Alternatively phrased, a method has stronger dependencies on another class compared to its own. Generally this translates to the method accessing a lot of data (attributes) of another class. In terms of software design, this smell implies that the method is strongly *coupled* to another class and therefore in the wrong location. The refactoring strategy is to simply move the method to the envied class. A basic example of Feature Envy is shown in Listing 2.1. Here, the `getAddressLabel()` method is smelly and should be moved from the `Person` to the `Address` class.

Listing 2.1: Example of Feature Envy.

```java
class Person {
   private Address address;
   [...]

   // Smelly method, envies 'Address' class.
   public String getAddressLabel() {
      return address.getStreetAndNumber() + "\n" +
             address.getZipCode() + " " +
             address.getResidence();
   }
}
```

Fowler admits that many cases are not straightforward in practice. Sometimes only part of a method is envious. It is also possible that a method envies several classes, to varying degree. Finally, a method may have many dependencies on its current class, in addition to strongly envying another. All of these non-trivial cases are more likely for methods with larger method bodies. Fowler suggests to split such methods into smaller ones and to reassess them separately.

Part of the Feature Envy definition is abstract. Detection of this smell requires a method to determine 'envy', originating from a method towards a class. Given a measure for envy, one could then compare the envy of a method towards its its parent and other classes.

2.2.2 Large Class

Affects classes. The Large Class is an OOP class that has become too bloated over time. Such large classes are generally hard for developers to understand and work on. This is due to limitations in human cognition, which is heavily taxed when reading and understanding source code. In addition, large classes often break the *single responsibility principle*. This design principle states that any given class should only have one reason (responsibility) for which it needs to be changed [16, ch. 8]. The reasoning behind this is that application responsibilities should be *decoupled*, so changes to one responsibility within a class cannot unexpectedly break others. A related concept is *cohesion*, the degree to which the elements inside a class belong together. The Large Class is refactored by separating off attributes and methods into smaller classes.

Part of the Large Class definition is abstract. Fowler specifies no specific metric(s) nor threshold values for which to consider classes as being too large. According to SourceMaking [23], signs are whenever a class contains too many fields, methods, or lines of code. This is however still vague.

2.2.3 Long Method

Affects class methods. Similar to the Large Class smell, a Long Method is a method that has become too bloated. In terms of software design, long methods are also generally hard for developers to understand and maintain. The refactoring strategy is to decompose the method into smaller methods, which can then be invoked from the original method. A given heuristic is that whenever a block of code in a method could use commenting, it should be extracted instead. If extracted methods are named after their purpose, developers may not even need to read their bodies while seeing the original method. An example of a refactored Long Method is shown in Listing 2.2.

Part of the Long Method definition is abstract. Fowler specifies no specific metric(s) nor thresholds for which to consider methods as being too large. SourceMaking states that methods containing more than 10 lines of code should start raising concerns [23].

Listing 2.2: Example refactoring of Long Method.

```
class Foo {
   public void aRefactoredLongMethod() {
      queryDatabase(..);
      displayResults(..);
      logResults(..);
   }

   // Extracted smaller methods
   public void queryDatabase() { .. }
   public void displayResults() { .. }
   public void logResults() { .. }
}
```

2.2.4 Message Chain

Affects method invocation expressions. A Message Chain occurs whenever code requests an object for another object, which it then asks for another object and so on. An example of such a method invocation chain is shown in Listing 2.3. Here, a message is sent to an instance of class A, which returns an instance of class B which is then sent the actual desired message.

This smell indicates that the chain of method invocation expressions (and therefore its parent method and class) is too strongly coupled to the modelled relationships between classes. Changes to any relationship would require all involved message chains throughout the program to be updated as well. This makes it harder for developers to evolve software. A design principle related to this smell is the *Law of Demeter*, which states that classes should only communicate with their immediate relational neighbours. Refactoring this smell is done by adding delegation methods to break up the chain. An example of this is shown in Listing 2.3.

Listing 2.3: Message Chain example.

```
// This smelly chain reaches 'through' class A to class B.
instanceOfClassA.getClassB().doSomething();

// --- After refactoring ---
instanceOfClassA.doSomething(); // chain is broken

class A {
   public void doSomething() { // delegation method, hiding class B
      self.getClassB().doSomething();
   }
}
```

A straightforward metric which can be used for smell detection is the chain length of method invocation expressions. Fowler specifies no particular thresholds, therefore the 'correct' one is open to interpretation.

2.2.5 Spaghetti Code

Affects classes. As the exception, Spaghetti Code was not defined by Fowler but instead introduced as an *anti-pattern* by Brown et al. [6]. Anti-patterns are bad solutions to common software design challenges. This is similar to the concept of code smells. However, while code smells are only *indications* of design issues, anti-patterns are universally bad. Within this thesis, Spaghetti Code was treated as another code smell.

Spaghetti Code occurs when an OOP class is implemented in a *procedural* manner. In procedural programming languages, methods (called procedures) operate on data structures that are part of a global program state. Key differences compared to OOP are the complete decoupling of data from methods and the absence of classes. Even though Java is object-oriented, it is still possible to misuse Java classes in a procedural manner. This may occur with developers who are not yet skilled with object oriented code. Spaghetti Code classes are hard for developers to understand because they lack the object-oriented structure that is normally expected.

Brown et al. supplied a very broad description of symptoms for Spaghetti Code. A more concise description was given by Moha et al. [17]: "Spaghetti Code is revealed by classes with no structure, declaring ***long methods*** with no ***parameters***, and utilising ***global variables***. ***Names*** of classes and methods may suggest ***procedural*** programming. Spaghetti Code does not exploit and prevents the use of object-orientation mechanisms [. . .]."

Note that this represents procedural programming on an object oriented class level. Methods are written as if they were procedures and class attributes are used as global state. In addition to understanding such classes, maintaining them can also become problematic. As attributes are procedurally accessed and modified from multiple methods, changing one method may unexpectedly affect others within the class.

2.3 Measuring Detection Performance

The concept of measuring detection performance enables the validation and comparison between different code smell detection techniques. A code smell detector may include the ability to detect multiple code smell types, each with their own detection performance. Two mainstream performance metrics are *precision* and *recall*.

The task of detecting code smells can be modelled as a *binary classification problem*. This is where a dataset of instances is given, for which each instance must be classified as either positive or negative. For code smell detection, the instance dataset consists of software components that can be affected by a single code smell. A code smell detection technique effectively classifies each component as being either smelly or non-smelly.

Precision and recall are two important metrics used in binary classification. In this context, they are calculated using: (1) the dataset of software components, (2) the set of components classified as smelly and (3) the set of components actually affected by the code smell. These three sets are drawn in Figure 2.1. The calculations for precision and recall are:

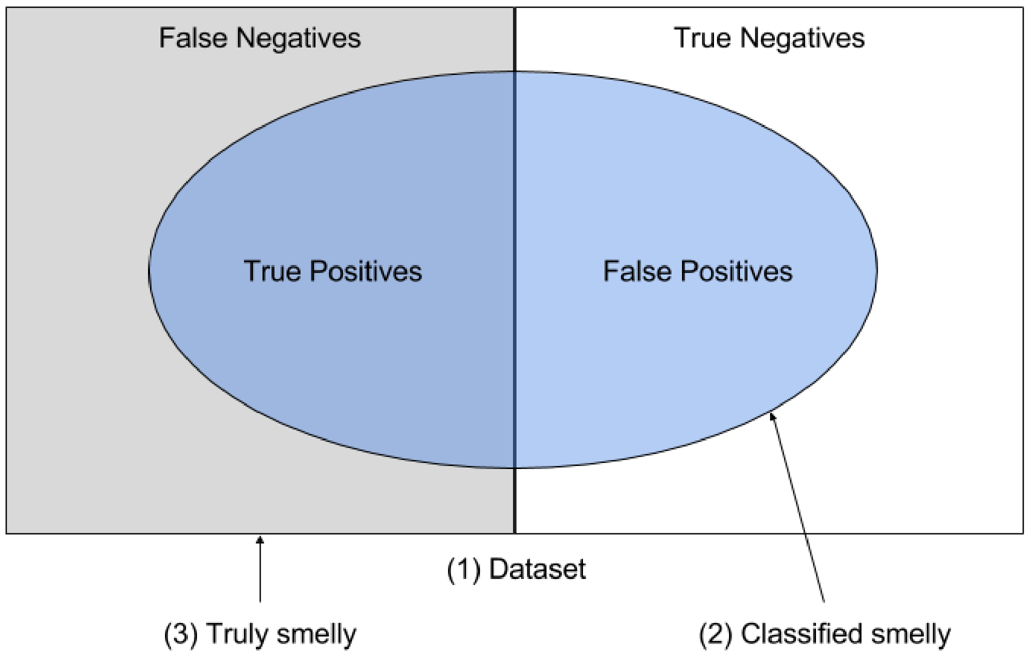

Figure 2.1: Steps to calculate precision and recall.

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}} \tag{2.1}$$

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}} \tag{2.2}$$

Both metrics assume values in the range $[0,1]$. Precision is the fraction of smells returned by the detector that are truly smelly. Recall is the fraction of true smells in the code that are being found. It is desirable for a detection technique to have both precision and recall as close as possible to 1.

A recurring pattern in code smell detection techniques is the use of metrics that correlate with a code smell, combined with a chosen threshold value to classify smelly instances. The threshold value can be varied in such cases and is directly related to the precision and recall of the technique. A stricter threshold with a smaller detection range is less likely to produce false positives (improved precision) but more likely to produce false negatives (worse recall). The reverse holds for a less strict threshold.

2.4 Android Platform Overview

This section briefly describes the Android platform. The focus is on those aspects that are specific to Android and are relevant to the detection of code smells.

2.4.1 Integrated Development Environment (IDE)

Android apps are written using Android Studio [3], a dedicated IDE from Google. The first stable release of Android Studio was in December 2014. Before this time, apps were developed using the Eclipse IDE [8] and the Android Development Tools (ADT) plug-in from Google. ADT was eventually deprecated in June 2015. Due to a transition period, it is possible to encounter Android projects that have not yet migrated to Android Studio.

2.4.2 Software Development Kit (SDK)

Android Studio exposes two different development kits using distinct programming languages. The Android SDK is written in Java, while the Android Native Development Kit (NDK) is written in C/C++. Developers use these respective languages to program against a development kit. In general, code smell detectors are created for a specific programming language. The existence of Android source code in two languages could have complicated matters with respect to running existing detectors.

However, the Android documentation states that the NDK is only meant for a particular set of use-cases and normally not needed [1]. Examples of use-cases are apps that demand real-time computing (such as graphically intensive games) and apps that require the use of legacy C/C++ libraries. Given this information, the scope of this research was adjusted to investigate Android apps solely using the Android SDK. As a result, only smell detection tools for Java needed to be considered.

2.4.3 Android App Architecture

The Android SDK is fully documented [2]. It is also fully object-oriented. Developers extend from building blocks supplied by the SDK to create their own applications. Some frequently used object classes are:

- **Activities** (`android.app.Activity`): An activity represents "a single screen with an user interface (UI)". For example, an e-mail app may have separate activities for: (1) browsing a list of messages, (2) viewing a single message and (3) composing a new message. As a user enters and navigates through an app, activities involved are continuously being created and destroyed. These (and more) states and transitions between them represent the *activity lifecycle*.

  For each new screen in an Android app, developers create a class that inherits from the `Activity` class. This class contains many placeholder methods which are invoked throughout the lifecycle of the activity. These are methods such as `onCreate()`, where an activity is expected to initialize its user interface. Through method overriding, custom behaviour is achieved.

- **Fragments** (`android.app.Fragment`): A fragment assumes the responsibilities of an activity for a portion of the user interface. More complex interfaces can consist of multiple 'panes', either side-by-side or overlapping. Instead of managing the whole

UI, an activity can delegate parts to one or more fragments. The relationship between activities and fragments is highlighted in Figure 2.2.

- **Views** (`android.view.View`): While activities and fragments manage the user interface, the actual elements visible on the screen are represented by views. The `View` class forms the basis of many subclasses for displaying buttons, sliders, drop-down lists and more. Views are laid out in a tree hierarchy, managed by their parent activity or fragment.

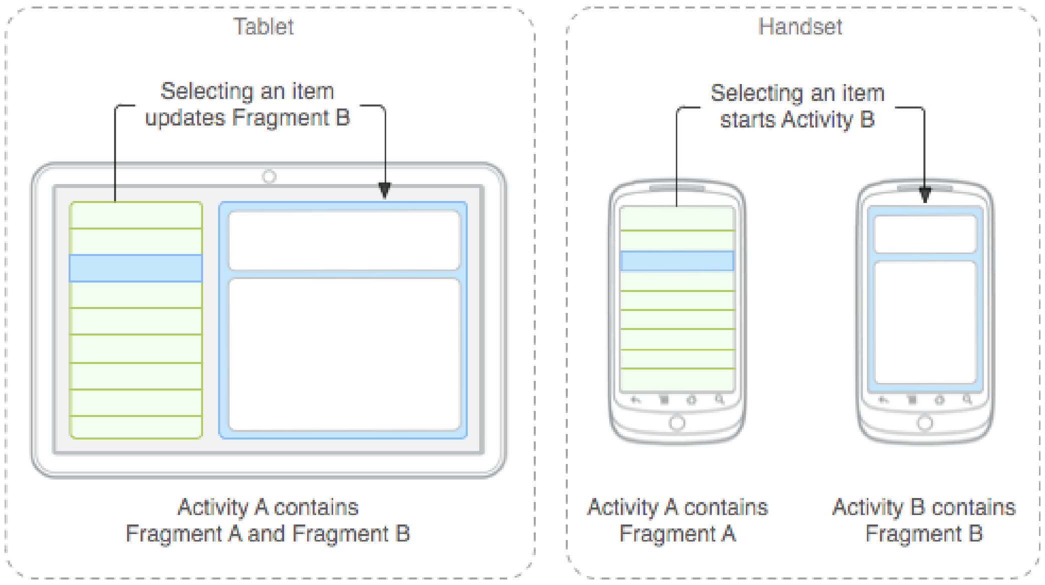
Tablet
Selecting an item updates Fragment B
Activity A contains Fragment A and Fragment B
Handset
Selecting an item starts Activity B
Activity A contains Fragment A
Activity B contains Fragment B

Figure 2.2: Android Activities and Fragments. Image from SDK documentation.

2.5 Related Work

Related work can be divided into (1) proposed smell detection techniques and tools, (2) empirical studies on the results between detection tools.

2.5.1 Techniques and Tools

Feature Envy Atkinson and King [4] proposed a basic technique to, within each method body, track the distinct field references towards classes other than its own class. An external class with many referenced fields was deemed a location to move the method to. By setting a *threshold* value, methods could be accepted or rejected as smelly. The authors created the Look# tool for the C# language that implemented this technique, among others.

A more elaborate technique was proposed by Tsantalis and Chatzigeorgiou [24]. They introduced the notion of an *entity set* corresponding to an entity (attribute/method/class): the set of entities that is used by or uses the entity. For methods, this was the set of attributes and methods accessed from it. For classes, this was the set of attributes and methods declared in it. An additional list of rules described cases in which certain elements should be excluded from the entity set. The *Jaccard similarity coefficient* between method- and class entity sets was then used as a metric for envy. This technique was integrated in the *JDeodorant* plug-in for Eclipse and Java.

While the first two techniques use the structural information of source code, a different approach is to analyse the way identifiers are used [5][20]. This *lexical* approach attempts to group fields and methods based on the names developers have chosen for them. The derived *clustering* is then compared to the current situation to suggest method refactorings. For Java, *TACO* [20] is a textual-based smell detector.

Large Class DECOR [17] is “a method that embodies and defines all the steps necessary for the specification and detection of code [...] smells”. It is a framework in which code smells are described using ‘rule cards’. Each card contains a *logical conjunction* of code metrics and corresponding thresholds. The metrics used for Large Class included the number of dependencies on data classes, LCOM5 (Lack of Cohesion Of Methods), the sum of declared attributes and methods, and the presence of certain keywords in the class name.

An indirect approach is to focus on opportunities to apply the *Extract Class* refactoring, instead of finding the smell itself. Note that the detection of attributes and methods that should be extracted from a class also detects instances of the Large Class. The focus here is on class cohesion. Fokaefs et al. [10] used a *clustering algorithm* on the attributes and methods of a class. For the distance between elements the *Jaccard distance* was used on their corresponding entry set. The notion of entity sets was reused from Tsantalis and Chatzigeorgiou [24].

Long Method The traditional method to measure method length is to count the number of lines of code (LOC) in the method body. However, this metric does not account for cases in which a large LOC is justified. For example, while a method containing a long switch statement with short cases has large LOC, breaking it up is a questionable refactoring.

Existing detection techniques mostly use the LOC metric. DECOR and PMD [22] both use a threshold value of 100. Checkstyle [7] uses a threshold value of 150. Tsantalis and Chatzigeorgiou [25] implemented a more elaborate technique into JDeodorant. They used the *control flow graph* corresponding to method code to extract code *slices* in which a variable is either being computed or updated. After passing a set of rules, such slices were proposed to be eligible for the *extract method* refactoring. Note that such proposals contain detections of the Long Method smell.

The *TACO* [20] textual-based smell detector is capable of detecting this smell. It finds a Long Method whenever it detects sets of statements in a method that are semantically distant from each other.

Message Chain A basic technique to detect this smell is to find instances of consecutive (chained) method invocations in the code. By definition, a method invocation in a 'chain' of length one cannot be affected by Message Chain. Finding consecutive invocations is possible by traversing the *Abstract Syntax Tree* (AST), a tree representation of typed code. However, returning all chains with length ≥ 2 is very lenient, returning the maximum amount of false positives. Stench Blossom [18] is a tool that returns the 'strength' value of the Message Chain smell based on the chain length.

DECOR uses the number of transitive invocations (NOTI), which is the number of consecutive method invocations through different classes. The rationale behind this metric was to ignore instances of consecutive invocations on the same class. An example of this is the chain `aString.trim().length()`, where the quality issue implied by the code smell is non-applicable. The NOTI threshold used in DECOR is 4.

Spaghetti Code The DECOR rule card for Spaghetti Code is based on the summary of the smell made by the authors. This summary was given in Section 2.2.5. For reference, the rule card used by DECOR is shown in Figure 2.3. The rule on line 2 is the intersection of the six rules lines 3-8.

```
1   RULE_CARD:SpaghettiCode  {
2     RULE:SpaghettiCode
            { INTER LongMethod  NoParamete  NoInheritance
            NoPolymorphism  ProceduralName  UseGlobalVariable };
3     RULE:LongMethod       { METRIC LOC_METHOD VERY_HIGH 10.0 };
4     RULE:NoParameter      { METRIC NMNOPARAM VERY_HIGH 5.0 };
5     RULE:NoInheritance    { METRIC DIT 1 0.0 };
6     RULE:NoPolymorphism   { STRUCT NO_POLYMORPHISM };
7     RULE:ProceduralName   { LEXIC CLASS_NAME
                                    (Make, Create, Exec...) };
8     RULE:UseGlobalVariable { STRUCT USE_GLOBAL_VARIABLE };
9    };
```

Figure 2.3: DECOR rule card for Spaghetti Code.

2.5.2 Empirical Studies

Fontana, Braione and Zononi (2012)[11] explored the spectrum of smell detection tools at the time for Java. This produced seven detection tools: Checkstyle, DECOR, inFusion, iPlasma, JDeodorant, PMD and Stench Blossom. Each of these tools support the detection of at least one of the five code smells investigated in this thesis. Three tools were dropped in the experimental phase. iPlasma was dropped being the predecessor of inFusion, while DECOR and Stench Blossom did not produce results that were easily exported.

The results of their experiments suggested that the four remaining tools (Checkstyle, JDeodorant, inFusion and PMD) did not meaningfully agree in their results. While the tools were individually useful, the authors suggested that developers use more than one tool for any given code smell.

Chapter 3

Constructing a Smell Oracle

For the two studies performed in this thesis, we needed the ability to measure the detection performance for a given technique or tool. As described in Section 2.3, performance can be expressed in terms of precision and recall. These metrics were calculated over a given dataset of source code, by comparing the set of *candidate* code smell detections from a technique/tool with the set of *actual* smell instances. Given a fixed dataset of Android source code, these metrics allowed direct comparisons between different techniques and tools.

However, the set of actual smell instances in Android code was not trivial to obtain. In general, such data does not inherently exist for any source code project. In addition, there would be no need to research code smell detection if an automated program existed that could perfectly produce actual smell instances. Regardless, actual smell instances still needed to be obtained in some way. The 'truth provider' of this information will be referred to as the *smell oracle*. This is named after the concept of the *oracle machine*: an abstract machine capable of perfectly solving complex, even non-computable problems.

The solution for obtaining a smell oracle was to manually create one. Given the definitions of the five investigated code smells in Section 2.2, a selection of Android app projects was manually inspected for code smells. These manually detected smells were then used to form the smell oracle, with its 'coverage' limited to exactly the source code of the selected Android apps. Although this procedure was valid in theory, there were some considerations. Firstly, the manual inspection of large quantities of code was expected to be very time-consuming. Secondly, the validity of this procedure and the resulting smell oracle was strongly dependent on correct judgement during manual code detection. After consideration, it was determined that there was no other viable way to obtain an oracle. As an aside, the practice of using human judges has been used before in research to validate the output of novel detection techniques.

This chapter covers the construction of the smell oracle. Section 3.1 describes the procedure for selecting and gathering the Android apps that were included in the dataset. Section 3.2 describes a preprocessing stage that was applied to the dataset in order to reduce the amount of manual work required. Section 3.3 describes the manual smell detection process, while Section 3.4 discusses the completed oracle.

3.1 App Selection

It was infeasible to gather and manually inspect source code for the complete universe of Android apps. Therefore, a selection of apps was made. In consultation, it was decided that at least five Android apps should be included in the source code dataset and smell oracle. Several selection criteria were used to find apps that were considered suitable.

The Android apps needed to be (1) open-source. This ensured that the corresponding source code of the apps would be freely available. The apps also needed to be (2) relatively small in source code quantity. This criterion was judged using the total number of classes en methods of the app. As mentioned earlier, manual inspection of source code was expected to be very time-consuming. Limited time resources dictated the selection of either more smaller apps or fewer larger apps. Selecting (even) fewer apps risked obtaining a distribution of code smells that was biased towards the coding style and experience of fewer developer teams. Although the validity of the smell oracle would not be affected, the intent was for the dataset of Android code and derived performance metrics to generalize as much as possible. This made the selection of smaller apps more desirable. Finally, the apps should be (3) popular in use. This made the dataset of Android code more significant and useful. This criterion was judged using the number of app user ratings in the Play Store, Google's dedicated application store.

The Play Store itself was potentially a large source of Android apps to select from. However, it would not have been trivial to randomly select apps conforming to criteria (1) and (2), as the necessary information is not listed in the store. Palomba et al. [21, Table I] proposed an automated approach to link the content (bug reports, feature requests) in user app reviews back to the involved software components. Although this research was not directly related to the thesis, the set of 10 Android apps involved proved to be useful. These apps all passed criteria (1) and (3), with criterion (3) being satisfied given the presence of least 10.000 user ratings on the Play Store.

Table 3.1: Android apps included in the smell oracle.

App	Description	Classes	Methods	User ratings
Cool Reader	E-book reader	114	1721	236.161
Focal	Camera app	81	839	12.413
SMS Backup Plus	Backup utility	80	557	57.864
Solitaire	Game	30	357	116.058
WordPress	Blogging app	469	5656	98.383
Amaze File Manager	File manager	140	1247	9.196

From these 10 apps five were selected conform criterion (2), listed in Table 3.1. These apps contained a small amount of classes and methods *relative* to the original 10. It should be noted that no previous empirical research was found exploring distributions of size metrics in Android projects. Although Java projects are a superset of Android projects and represent a broader research domain, any thresholds classifying small/medium/large projects would not necessarily translate to Android.

An important consideration with respect to app selection was the estimated size of the resulting smell oracle. With an oracle containing relatively few smell instances, the accuracy and significance of derived detection performance metrics would be negatively affected. This notion held for each of the five code smells under investigation. Given our initial app selection, we projected that Spaghetti Code would only have few smell instances. For this reason Amaze File Manager was retroactively added as a sixth app in order to broaden the scope and collect more ‘data’. This app was found by browsing F-Droid [9], a repository of open-source Android apps, until an app with size metrics similar to the original five was found.

3.2 Preprocessing

The source code of the apps selected in Section 3.1 was downloaded via the F-Droid repository [9]. Even though the apps were intentionally chosen to hold back the quantity of source code, it became clear that the number of classes and methods in Table 3.1 was still too large to manually inspect. Therefore, a preprocessing stage was used to reduce the quantity of code that would have to be inspected. A diagram of the extended process used to create the smell oracle is shown in Figure 3.1.

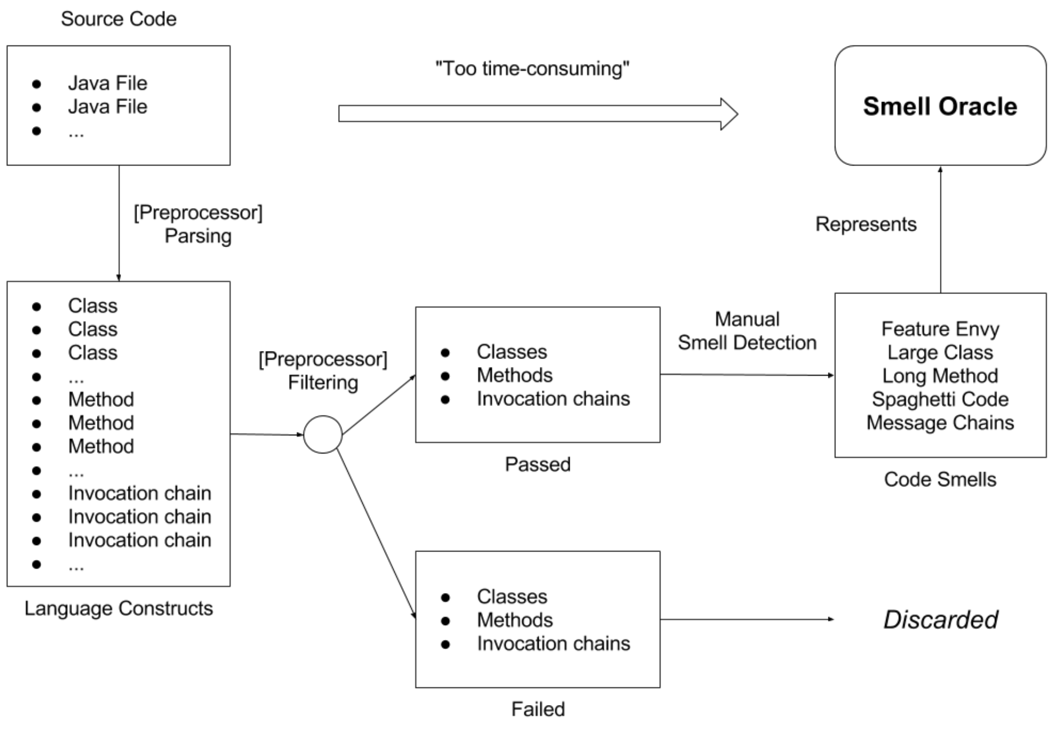

Figure 3.1: Extended process used to create the smell oracle.

The first preprocessing stage was to parse all relevant language constructs from the source code. The five code smells under investigation affected either classes, methods or method invocation expressions. Instead of manually traversing a folder hierarchy of Java files, it would be easier to extract just these constructs. This produced a dataset of Java language constructs. Note that with respect to the original source code these constructs may overlap. For example, the code from a parsed method will also be present in its parsed parent class. Through parsing, it was possible to shift from having to inspect many source code files to the simpler task of iterating through a list of language constructs. An overview of the parsing results is shown in Table 3.2.

Table 3.2: Number of language constructs parsed in preprocessing.

Constructs:	Classes	Methods	Invocations
Amaze File Manager	187	1247	10.327
Cool Reader	214	1721	8577
Focal	125	839	3856
SMS Backup Plus	97	557	2394
Solitaire	30	357	1411
WordPress	728	5656	29.940
Total	1381	10.377	56.505

The quantities involved in Table 3.2 were still far too large to manually inspect. The second preprocessing stage filtered out *trivial instances* from the set of parsed language constructs. Trivial instances are defined as being very unlikely to be affected by a code smell, based on one or more code metrics. By removing these instances without the use of a human expert, the number of instances to manually inspect was significantly reduced. Filtering was used carefully and was intended not to affect the validity of the smell oracle. In theory, filtered out constructs would have been removed in the same way if inspected manually. However, this was dependant on the filtering stage not accidentally removing smelly instances.

Both parsing and filtering were implemented in a *preprocessor tool*. This tool was written in Java and is publicly available on GitHub[1]. For parsing, the Eclipse IDE provided the Eclipse Java Development Tools (JDT), a framework which helps with parsing Java code. For filtering, *filtering rules* were applied to parsed language constructs, for each of the five code smells. The rules used are listed in Table 3.3. They were based on the definitions of the corresponding code smells, as covered in Section 2.2. Explanations for the choice of filtering rules are supplied later in this section. An overview of the output of the preprocessor is shown in Table 3.4. It breaks down the number of language constructs left for manual inspection, for each code smell and each Android app.

The output of the preprocessor produced quantities that were almost viable for manual inspection. The specific combinations of WordPress-MessageChain and WordPress-FeatureEnvy left relatively many instances compared to other combinations. Completely

[1]`https://github.com/DustinLim/mscthesis-preprocessor`

Table 3.3: Filtering rules used in preprocessing.

Code smell	Affects	Passes through filter iff:
Feature Envy	Methods	Another class exists for which this method has larger envy (Equation 3.1) than towards its own class.
Large Class	Classes	Body contains $>$ 438 lines of code (LOC).
Long Method	Methods	Body contains $>$ 80 LOC.
Spaghetti Code	Classes	Body contains $\geq$ 2 methods that have $>$ 80 LOC.
Message Chains	Invocation chains	Chain has length $\geq$ 2 and traverses $\geq$ 2 different declaring classes.

Table 3.4: Overview of preprocessor results.

	Feature Envy	Large Class	Long Method	Message Chain	Spaghetti Code
Parsed constructs					
From all apps	10.377	1381	10.377	44.511	1381
Construct type	Method	Class	Method	Invocation chain	Class
Filtered constructs					
Amaze File Manager	146	15	26	208	5
Cool Reader	329	15	18	163	7
Focal	90	7	4	124	0
SMS Backup Plus	73	1	1	56	0
Solitaire	94	2	2	26	0
WordPress	(901)*	34	36	(650)*	6
Total Remaining	732	74	87	577	18

* = Combination dropped from smell oracle.

discarding WordPress from the dataset would have significantly reduced the number of instances left for Large Class, Long Method and Spaghetti Code. As a compromise, WordPress was discarded for just the Feature Envy and Message Chain smell. With respect to the validity of the oracle, this posed no issue as long as the same combinations were ignored for determining precision and recall.

The remainder of this section (3.2.1-3.2.5) justifies and describes the implementation of each of the filtering rules listed in Table 3.3.

3.2.1 Filtering Rule: Feature Envy

Rule reasoning Recall that Feature Envy occurs whenever a method accesses the data of another class more than its own parent object. This abstract description was turned into a filtering rule using a metric for envy.

A basic envy metric was chosen. In Java, a method can access entities outside of its declaration in two ways. Firstly, *field access* expressions directly access data attributes of an object. Secondly, *method invocation* expressions invoke the functionality of an object. These invocations can be either on accessor methods (getters and setters) or other methods. Accessors expose access to data similar to direct field access. In the case of non-accessor methods, note that methods generally perform their functionality using class data. Therefore, method invocations were considered as an indirect form of data access. Combining the above, the following metric was created for the envy between a method m and class c:

$$\text{Expr}_t(m,c) = \text{Set of expressions in method } m \text{ of type } t\text{, which bind to class } c.$$

$$\text{Envy}(m,c) = |\text{Expr}_{\text{FieldAccess}}(m,c)| + |\text{Expr}_{\text{MethodInvocation}}(m,c)| \tag{3.1}$$

The filtering rule in Table 3.3 was based on this metric. It was chosen as the least discriminating rule conforming to the definition of Feature Envy. If no classes exist for which a method has larger envy than towards its parent class, it is not affected by the smell.

Implementation The Eclipse JDT framework was used to parse the *Abstract Syntax Trees* (AST's) from Java files. An AST is a tree graph where the nodes represent language constructs and edges model their hierarchy. A basic snippet of Java code and its corresponding AST is shown in Figure 3.2. For entire Java files, parsed AST's are more complex, containing many depth levels for classes, methods, statements, expressions, etc. In addition, the Eclipse JDT framework also offered resolving of *bindings*. A binding is a relation between two AST nodes, potentially in different trees. An example is the binding of an identifier node (containing a string name) to its corresponding declaration node in another location.

The filtering rule was implemented by collecting `MethodDeclaration` nodes from all parsed AST's. For each method declaration, descendant nodes of type `FieldAccess` and `MethodInvocation` were collected. Both node types contained bindings to their respective `FieldDeclaration` and `MethodDeclaration` node. After following these bindings, the final step was to visit their ancestor `ClassDeclaration` node. This provided a collection of all points of data access within a method, grouped by class. This was sufficient information to apply Equation 3.1.

Caveats In practice, it was not possible for all field access and method invocation constructs to contribute to Equation 3.1. This is due to the possibility of unresolved AST bindings. Given a source code folder targeted by the preprocessor, the Eclipse JDT framework would be unable to resolve bindings towards destinations not located there. This issue occurred with bindings that would have resolved to classes in external libraries.

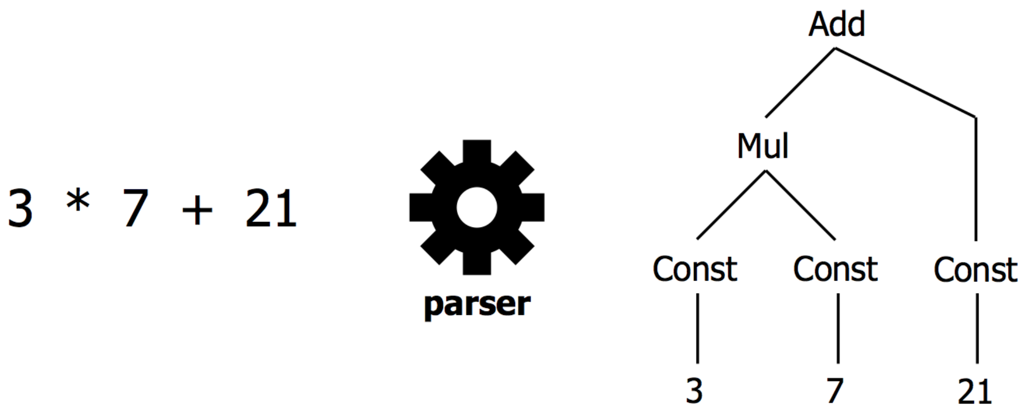

Figure 3.2: Java code snippet and its corresponding AST.

Note that the Feature Envy refactoring procedure of moving a method towards an envied class is not applicable for library classes not “owned” by the Android app developer. Developers not owning their external libraries is generally the case in software development. As a result, it was deemed appropriate to ignore any unresolved bindings with respect to Equation 3.1.

3.2.2 Filtering Rule: Large Class

Rule reasoning Recall that the Large Class occurs when a class is too bloated in size. A traditional size metric which is often used is the number of Lines Of Code (LOC). It was reasoned that classes with ‘normal’ LOC would be safe to filter out. Indeed, such classes would appear to contradict the definition of the Large Class. A boxplot of the distribution of class LOC on the selected Android apps is shown in Figure 3.3. The *upper outer fence* (defined as Q3 + $3 * \text{IQR}$) is not drawn in the boxplot but lies at 438. By definition, classes with higher LOC were *extreme outliers*. By setting the class LOC threshold equal to the upper outer fence, only extreme outliers passed the filter.

Implementation The implementation of the LOC metric was unexpectedly non-trivial. If the number of plain text lines were to be used directly, this would also count white- and comment lines. Both are meant to improve code presentation and readability; it would be inappropriate to count them towards the size of the class body. A second complication concerned code formatting. Differences in coding style and experience between developers may produce the same code using a different number of lines. An example of coding style is whether curly braces appear on a separate line. The LOC metric should be invariant to these issues.

The Eclipse JDT offered a solution. From a parsed AST, it was possible to reverse the operation and obtain Java code again. As differing coding styles would parse to the same AST, this reverse operation would produce ‘prettyprinted’ code with consistent formatting. It would also exclude code comments due to the way they were parsed into the AST, not

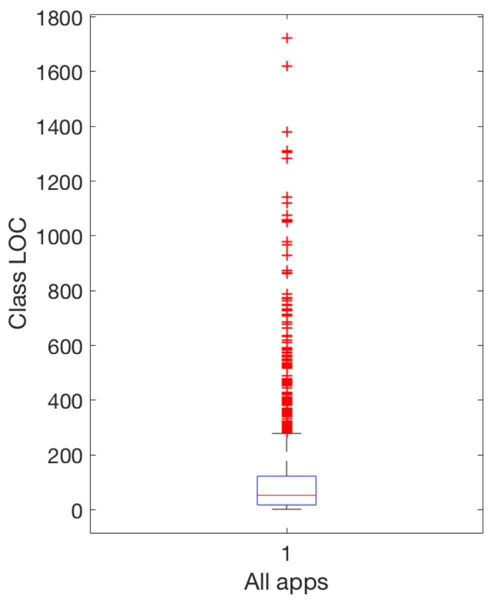

Figure 3.3: Boxplot of class LOC in selected Android apps. One extreme outlier is not shown.

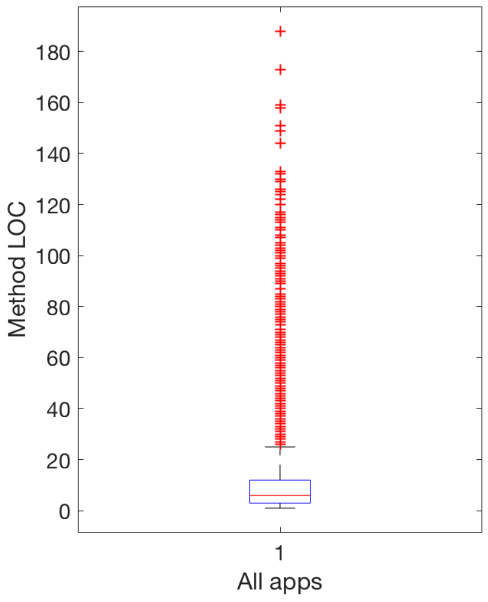

Figure 3.4: Boxplot of method LOC in selected Android apps. Two extreme outliers are not shown.

being actual code. For the LOC metric, text lines were counted from the prettyprinted code instead of the original code.

3.2.3 Filtering Rule: Long Method

Rule reasoning Recall that the Long Method smell occurs whenever a method is too bloated in size. Similar to the Large Class filtering rule, methods with normal LOC were filtered out. A boxplot of method LOC on the selected Android apps is shown in Figure 3.3. A threshold value of 80 lines of code was used for the filtering rule, located around the 99th percentile.

Existing smell detection tools as described in Section 2.5 have used threshold values of 100 and 150. Considering that the filtering rule was not meant to classify the smell but merely remove unlikely instances, the filtering threshold was chosen lower than existing detection thresholds.

Implementation Implementation was similar to the implementation for the Large Class filtering rule. From a parsed `MethodDeclaration` AST node, prettyprinted Java code was obtained with consistent formatting and without comments. The LOC metric was calculated by counting lines from this textual representation.

3.2.4 Filtering Rule: Message Chain

Rule reasoning The Message Chain smell occurs whenever a chain of method invocations traverses through multiple classes. The filtering rule chosen was a direct translation from this definition. Each method invocation had a corresponding class declaring the method. By definition, chains that are only length one or that do not traverse multiple declaring classes could not be affected by Message Chain.

Implementation The AST structure of an invocation chain is shown in Listing 3.1. It is a *path* of successive `MethodInvocation` nodes. The first step for the implementation was to collect all method invocation nodes from the AST's. Next, this set was transformed into the set of invocation chains in the source code. This was done by discarding all `MethodInvocation` nodes that had another method invocation node as its parent. This left only those nodes that were the root of their respective chain.

Chain length was determined using the number of successive `MethodInvocation` nodes from each root. Next, the number of declaring classes traversed was determined by following the binding from each `MethodInvocation` to a `MethodDeclaration` node and inspecting its ancestor class. For reference, Table 3.5 shows the frequency table of traversed classes over all invocation chains that were parsed.

Listing 3.1: AST structure of an invocation chain

```
// MethodInvocation node structure
MethodInvocation(
    <expression node>,
    <method name>
)

// Example invocation chain of length two
variable.methodA().methodB();

// Corresponding AST, one chain contains multiple MethodInvocation nodes

MethodInvocation(
    MethodInvocation(
        "variable",
        "methodA"
    ),
    "methodB"
)
```

3.2.5 Filtering Rule: Spaghetti Code

Rule reasoning Recall that Spaghetti Code occurs when classes have "no structure, declaring long methods with no parameters, and utilising global variables. Names of classes and methods may suggest procedural programming [...]." A reasonable filtering rule was a

Table 3.5: Frequency table of traversed classes in invocation chains.

Traversed classes	Count	Percent
1	43284	97.24%
2	1202	2.70%
3	24	0.05%
4	1	0.00%

more relaxed version of this description: a class containing at least two or more long methods. Methods were considered long using the same filtering rule used for the Long Method (method LOC $>$ 80).

Implementation For each `ClassDeclaration` node, descendant `MethodDeclaration` nodes were collected. The Long Method filtering rule was then applied to each method node. A class would pass the Spaghetti Code filter whenever two or more long methods remained.

3.3 Manual Smell Detection

The preprocessor produced a parsed and filtered dataset of Java constructs from Android source code. Each construct represented either a class, method, or method invocation chain and was associated with one of the five investigated code smells. The final step for constructing the smell oracle was to manually inspect the filtered constructs. Each construct was either accepted or rejected as being affected by the associated code smell.

Workflow In total, 1475 code constructs were manually inspected. The preprocessor was written to produce its output files in the comma-separated values (CSV) format, a representation of tabular data. A simplified example of this output is shown in Table 3.6. Each row represented a code construct, with the columns listing construct properties. One property was a generated command line statement, which would open a text editor at the location of the code construct. Another property was a generated *unique identifier* (UID) for each code construct. The CSV format was easily converted into a spreadsheet, used to manually extend the dataset. In particular, a true-false judgement for the smelliness of each construct was added in the first column. More detailed analysis was written in the second column.

Table 3.6: Preprocessor output table used with manual smell detection.

Smelly?	Analysis	Android App	UID	Xed command *
		output	*output*	*output*

* = *xed* is the Xcode text editor invocation tool [28].

Methodology The validity of the smell oracle was dependent on accurate manual smell detection. The source code for each code construct was inspected and judged for the presence of a code smell according to the following criteria:

- The code construct matches the abstract and concrete description of the code smell.
- The application of the corresponding refactoring is appropriate.

The remainder of this section describes the manual detection process for each of the investigated code smells.

3.3.1 Feature Envy - Smell Detection

The preprocessor was extended to add the following properties for each of the methods:

- **Internal envy:** A list of field and invoked method names that resolved to the parent class of the method. The size of this list equals the envy value (Equation 3.1) of the method towards its parent class.

- **Envied classes:** A dictionary containing fully qualified class names as keys and lists of field and invoked method names as objects. Each list contains the names that resolved to the class name in the key. As these are envied classes, each list contains more elements than the internal envy list.
- **Ignored:** A list of field and invocation names that remained unresolved using Eclipse JDT. Refer to Section 3.2.1 - Caveats for the discussion behind this list.

At least one envied class was listed for each method. The presence of Feature Envy was judged by tracking the names in *Envied classes* back to the method body. If the entire method should be moved to the envied class, this was accepted as Feature Envy. If the names were concentrated in part of the method, an alternative consideration was whether that part of the method should be moved. This was also considered an accepted instance. Each accepted and rejected method was provided with a short analysis. A complete overview of this analysis can be found in Appendix A.1. Table 3.7 highlights a few notable recurring patterns.

Table 3.7: Notable Feature Envy analyses.

Analysis	Instances

[Rejected] Method delegates its invocation to another class.
Method delegation occurs whenever a method contains no actual logic, forwarding the invocation instead:
`public boolean isDonationSupported() {`
`  return mDonationService.isBillingSupported();`
`     // invocation delegated to DonationService class`
`}`
In this example there is zero envy towards the parent class, with one reference to the `DonationService` class. Moving the method is however not justified, given that method delegation is a valid software pattern. Its purpose is to hide the delegate class from clients using the parent class of the method, reducing *class coupling*. As an aside, the method delegation pattern is also the refactoring strategy for the Message Chain smell.
	94

[Accepted] Refactoring (part of) method appears appropriate. These methods, or part of them, should be moved to an envied class.	82
[Rejected] Envy is spread over multiple classes. In this case there was no clear refactoring towards a particular envied class. The classes most envied shared a similar number of references towards it. If this was not the case, the method could have been moved to the strongest envied class.	76

[Rejected] Method uses an utility class. External references are due to usage of the services of an utility class. Examples of utility classes are those facilitating threading, on-disk file management and math operations. Refactoring was not justified because the methods represent custom usage of a general-purpose class.	59
[Rejected] Envy is towards superclass of the class declaring the method. These methods fall outside of the scope of the Feature Envy smell. In object-oriented programming, fields and methods are inherited from any superclasses.	58
[Rejected] Method manages collection of envied class. These methods work with multiple instances (arrays, lists, dictionaries) of the envied class. It is not possible to move such methods to the envied class, given that it models a single instance.	56

3.3.2 Large Class - Smell Detection

The preprocessor was extended to add the following properties for each of the classes: lines of code (LOC), number of attributes (NOA) and number of methods (NOM). The LOC metric was used previously for the filtering rule, NOA and NOM were included as additional size metrics.

It became apparent that the inspection of class bodies from top-to-bottom was a difficult cognitive task. Especially with unfamiliar code, reading the class body was too granular of a strategy to detect a Large Class. Instead, a higher level class overview was used for manual smell detection. A *polymetric view* was generated for each class using the inFusion tool. Polymetric views are metrics-enriched visualizations of software entities and their relationships [15, p. 7]. The inFusion tool itself was commercial software that is no longer publicly available. In this case, an old copy of the software was retained from previous work.

Figure 3.5 outlines the polymetric view. Class methods and attributes are separated and grouped based on their Java access level. Method squares grow based on the number of lines in the body. Attribute squares grow based on frequency of use within the class. As an interactive tool, clicking a square displayed the corresponding code in an editor.

Using these polymetric views, a class was accepted if a subset of attributes and methods represented one of multiple class responsibilities. In that case, the subset should be extracted into a separate class. The complete analysis is shown in Appendix A.2, Table 3.8 highlights a few notable recurring patterns.

Anonymous classes Anonymous classes are nameless class instantiations either at an attribute declaration, or inline inside a method. An example is shown in Listing 3.3. The Android SDK incentivises frequent use of anonymous classes for implementing interfaces such as `View.OnClickListener` and `Handler.Callback`. Anonymous classes were counted

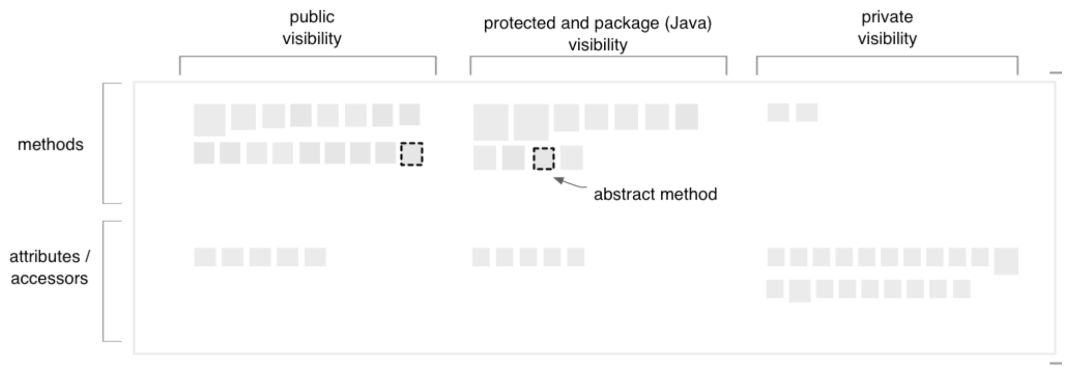

Figure 3.5: Polymetric class view from inFusion. Image taken from help files.

towards the size of their outer class. This is due to their purpose of executing a block of code at an later time.

Table 3.8: Notable Large Class analyses.

Analysis	Instances
[Rejected] Class size inflated due to inner class(es). In Java, an inner class can be declared inside the declaration of another (outer) class, as shown in Listing 3.3. It was decided not to count inner classes towards the size of their outer class. The reasoning behind the Large Class is that a class should not manage too much data or functionality. However, an inner class is in itself an encapsulation of data and functionality. Although inner classes could be extracted from their outer class, this would accomplish little in terms of code quality. These classes were no longer bloated when their their inner classes were ignored.	16
[Accepted] Multiple class responsibilities visible. These classes contained signs of containing multiple responsibilities. This was based on either the naming of attributes and methods, or the presence of comment blocks dividing the class body into sections.	16
[Rejected] No obvious refactoring. Even though these classes appeared large, it was unclear which attributes or methods should be refactored. A concrete example is the `GeneralDialogCreation` class from Amaze File Manager. This class contained static-only methods that displayed all possible dialogs.	12
[Accepted] Too many attributes. Classes with a clear excessive amount of attributes.	10
[Accepted] UI class managing too much UI. An Android-specific analysis. These are either Android *activities* or *fragments* that should delegate part of the UI to a separate class.	10

Total accepted instances:	43
Total rejected instances:	31

Listing 3.3: Example of an inner class and anonymous class.

```
class OuterClass {
   class InnerClass { [Attributes], [Methods] }

   public void anOuterClassMethod() {
      Button button = (Button) findViewById(R.id.myButton);

      // Declare anonymous OnClickListener to respond to button tap
      button.setOnClickListener(new View.OnClickListener() {
         @Override
         public void onClick(View V) {
            // Block of code
         }
      });
   }
}
```

3.3.3 Long Method - Smell Detection

The preprocessor was extended to add the LOC metric for each method. The complete analysis is found in Appendix A.3. Table 3.9 highlights a few notable recurring patterns.

Table 3.9: Notable Long Method analyses.

Analysis	Instances
[Accepted] Visible method section(s) to extract. Generally these methods have comments visibly highlighting sections of code to extract. An alternative observation was that many Android methods overridden by developers made the Long Method more apparent. Methods such as `Activity.onCreate(Bundle)` are points for developers to perform their initialization. It is easily visible whenever large sections of code initialize separate things, making refactoring straightforward.	18

[Accepted] Contains large inline anonymous class(es). Android uses an event system, by which developers can react to various events. These include touch events from the user and general system events. Frequently developers declare one or more anonymous classes within a single method to respond to multiple events. Although anonymous classes are not inherently bad, they can cause a bloated parent method. A method should perform a single task, while the anonymous classes handle various events. They can be refactored into inner classes, or can be declared as an attribute outside of the method.	17
[Rejected] Contains if-else or switch conditional with many small cases. Breaking down a conditional with many small cases did not generally reduce code complexity.	15
[Accepted] Contains if-else conditional with large cases. These methods were easily identified for refactoring. As the construct represents a decision point, the if/elseif/else cases are easily refactored into separate methods.	14
Total accepted instances:	58
Total rejected instances:	29

3.3.4 Message Chain - Smell Detection

The preprocessor was extended to print the following supporting information for each invocation chain: (1) the method names in the chain, for example

```
.getColorPreference( ).randomize( ).saveToPreferences( )
```

and (2) the resolved parent classes for each method in the chain:

```
[com.amaze.filemanager.activities.BasicActivity,
com.amaze.filemanager.utils.color.ColorPreference,
com.amaze.filemanager.utils.color.ColorPreference]
```

The complete analysis is in Appendix A.4, Table 3.10 highlights a few notable recurring patterns.

Table 3.10: Summary of manual Message Chains detection.

Analysis	Instances
[Rejected] Some invoked types are part of the Java/Android platform. Classes from the Java or Android platform can not be modified to shorten a message chain. Ignoring these invocations, the chain no longer traverses more than one class.	230

[Accepted] Hide Delegate applicable. The suggested refactoring for the smell can be applied.	84
[Rejected] Part of chain invokes local methods inherited from superclass. In object-oriented programming, inherited superclass methods are treated as being local.	57
[Rejected] Static method pattern is present. `Services.getScanner( ).getDownloadDirectory( )}.` Although this example chain passed through a services and scanner class, the first invocation was on a static method. This is a legitimate software pattern, exposing methods for global use.	50
Total accepted instances:	101
Total rejected instances:	465

3.3.5 Spaghetti Code - Smell Detection

Notably, all preprocessed classes were manually rejected for this smell. It was not easy to troubleshoot this result. The filtering rule used (class contains ≥ 2 methods that have > 80 LOC) was already a very relaxed version of the Spaghetti Code definition. Still, a possibility remains that smelly instances were erroneously filtered out.

A more likely cause is that our interpretation and judgement of Spaghetti Code was too strict for any of the classes to pass. To reiterate the description by Moha et al. [17]: "Spaghetti Code is revealed by classes with no structure, declaring ***long methods*** with no ***parameters***, and utilising ***global variables***. ***Names*** of classes and methods may suggest ***procedural*** programming. Spaghetti Code does not exploit and prevents the use of object-orientation mechanisms [. . .]." The same paper highlighted two concrete examples [17, p. 12] of Spaghetti Code in XERCES v2.7.0, a framework for building XML parsers in Java. Interestingly, the methods in these two examples do accept parameters, in contrast to the description of the smell. It appears that, at least in that research, classes were accepted as smelly if 'enough' of the definition was applicable as judged by a majority vote between their four experts. It was not clear for this research where to draw this line.

A final consideration is that there were simply no affected classes in the given dataset of Android code. It should be noted that the Android operation system is relatively young, initially being released in 2008. With Object-oriented programming being a mainstream programming paradigm by that time, it is possible that developers are less likely to write procedural code resulting in the Spaghetti Code smell compared to other software platforms.

For reference, Table 3.11 supplies a summary of the detection results.

Table 3.11: Summary of manual Spaghetti Code detection.

Analysis	Instances
[Rejected] The long methods have parameters.	6
[Rejected] Method override(s) of Android callbacks.	5
[Rejected] Utility class with static methods and little attributes.	3
[Rejected] Long method which updates UI.	3
[Rejected] Constructor long method.	1
Total accepted instances:	0
Total rejected instances:	18

3.4 Results

Table 3.12 gives an overview of the constructed smell oracle. The quantities of code constructs between each of the stages as described in Figure 3.1 is also included.

Table 3.12: Summary of constructed smell oracle.

Code Smell	Affects	Parsed	Preprocessed	**Included in Oracle**
Feature Envy	Methods	10.377	730*	84*
Large Class	Classes	1381	74	43
Long Method	Methods	10.377	87	58
Message Chains	Invocations	56.505	566*	101*
Spaghetti Code	Classes	1381	18	0

* = WordPress app not included.

Chapter 4

Study I: Using Existing Smell Detectors on Android Apps

The smell oracle constructed in Chapter 3 was used to explore the detection performance of existing code smell detectors on Android apps. In this context, the existing detectors were specifically for Java and designed without the Android platform in mind. The empirical study aimed to answer the following research questions:

- **RQ1**: How well do existing smell detectors perform on Android apps?
- **RQ2**: Are the smells detected by different detectors the same or are they different?

Section 4.1 describes the methodology used to answer the research questions. Section 4.2 presents the results obtained and corresponding analysis.

4.1 Methodology

The methodology consisted of a selection of smell detectors to include in the study, a method of converting of Android Studio projects into Eclipse projects and the generation of unique identifiers from smell detector results.

4.1.1 Included Detectors

Detectors were taken into consideration from the related work as covered in Section 2.5. Only Java detectors were suitable to run on Android code. Table 4.1 lists the detectors that were considered, including their supported code smells. In consultation, it was decided to include the highlighted detectors in the study. Together they covered all of the code smells in the smell oracle. Where possible, covering the same code smell with multiple detectors would enable answering **RQ2**. Notably, only one detector supported Spaghetti Code and only two supported Message Chain.

DECOR [17] is a conceptual framework in which code smells are described using ‘rule cards’. These cards contain a logical conjunction of code metrics and thresholds. A concrete implementation of DECOR and its rule cards was created by Palomba et al. [20], which was

Table 4.1: Considered Java smell detectors.

Detector	Feature Envy	Large Class	Long Method	Spaghetti Code	Message Chain
Checkstyle		×	×		
DECOR		×	×	×	×
inFusion	×	×			
iPlasma	×	×			
JDeodorant	×	×	×		
PMD		×	×		
Stench Blossom	×	×	×		×
TACO	×		×		
JSpIRIT	×	×			

Bolded = Included in study.

made available for this study. This implementation was in the form of an Eclipse project, in which an Android project path could be set for smell detection. Running the Eclipse project would then initiate a detection run. After each run, detections were printed to the console.

JDeodorant is a smell detector in the form of an Eclipse plug-in, meant to be used alongside development. To use it, an Android project needed to be loaded into Eclipse after which a detection run could be initiated. After each run, detections were presented in a dedicated view of the Eclipse IDE. JDeodorant v5.0.64 was installed from the Eclipse Marketplace and used in this study.

TACO [20] is a textual-based smell detector that analyses identifier names to determine code cohesion and detect related code smells. Similar to TACO, this detector was made available for this study in the form of an Eclipse project. After each run, detections were written to an output file.

JSpIRIT [26] is a smell detector in the form of an Eclipse plug-in. The detection strategies in it follow those presented in a book by Lanza and Marinescu [15]. Similar to JDeodorant in use, an Android project needed to be loaded into Eclipse. Detections were presented in a dedicated view within Eclipse.

As one of only two detectors supporting the Message Chain smell, Stench Blossom [18] was also considered for inclusion. Stench Blossom is an Eclipse plug-in which displays visual indicators in the presence of code smells within the code editor. However, this design made it infeasible to collect detection output without manually scrolling through each source file and writing down all detections.

4.1.2 Conversion of Android Studio Projects

The six Android apps included in the smell oracle were a mix of Android Studio and older Eclipse projects. This posed a challenge as it was not possible to initiate a detection run of JDeodorant and JSpIRIT within Android Studio.

While Google provides an automated migrator from Eclipse projects to Android Studio projects, the reverse was unsupported. For this study, reverse migration was done manually by modifying the Android Studio project folders into the correct structure for Eclipse. In addition, the *gradle-eclipse-aar* plugin [14], developed by a third party, proved essential. Android Studio packages library dependencies as `*.aar` files that are unreadable in Eclipse. The gradle-eclipse-aar plugin unpacked such aar resources into resources compatible with Eclipse.

4.1.3 Generating Unique Identifiers

The performance of detection tools was measured through precision and recall, as defined in Section 2.3. In this context, true positives are smell detections that are also present in the smell oracle. False positives are smell detections that are not present in the smell oracle. Finally, false negatives are instances in the smell oracle that remained undetected.

In order to determine true and false positives, instances in both (1) detector output and (2) smell oracle were translated into *unique identifiers*. This enabled the matching of instances between the two. Unique identifiers are strings that can represent a class, method or invocation chain in the code. They are unique in that different locations in Android app source code always translate into different identifiers. The two sets of identifiers corresponding to (1) and (2) were then compared. Unique identifiers were generated according to the following syntax:

- `FE-<appname>-<classname>.<methodname>@LN<line>`
- `LC-<appname>-<classname>`
- `LM-<appname>-<classname>.<methodname>@LN<line>`
- `MC-<appname>-<classname>.<methodname>@LN<line>:<column>`
- `SC-<appname>-<classname>`

Here, {`FE, LC, LM, MC, SC`} refers to code smell involved, `line` refers to the line number in the Java file and `column` refers to the character number on the line. Some examples of identifiers that were generated are:

- `FE-AmazeFileManager-AboutActivity.switchIcons@LN190`
- `LC-Cool Reader-FileBrowser`
- `MC-Focal-SettingsWidget.openWidgetsToggleDialog@LN443:16`

4.2 Results

The performance results that were obtained are shown in Table 4.2, broken down by code smell and detection tool. The number of true positives (TP), false positives (FP) and false negatives (FN) produced precision and recall. The F1 was also taken into consideration, which is defined as the *harmonic mean* of precision and recall. It reduces the two values to a single performance metric.

Table 4.2: Existing detector performance.

Detector	TP	FP	FN	Precision	Recall	F1 score
Feature Envy						
JDeodorant	10	30	74	0.25	0.12	0.08
JSpIRIT	25	425	59	0.06	0.30	0.05
TACO	8	580	76	0.01	0.10	0.01
Large Class						
DECOR	35	20	8	0.64	0.81	0.36
JDeodorant	25	140	18	0.15	0.58	0.12
JSpIRIT	33	105	10	0.24	0.77	0.18
Long Method						
DECOR	18	11	40	0.62	0.31	0.21
JDeodorant	0	0	58	NaN	0.00	NaN
TACO	43	275	15	0.14	0.74	0.11
Message Chain						
DECOR	9	94	92	0.09	0.09	0.04
Spaghetti Code						
DECOR	0	40	0	0.00	NaN	NaN

The detection of Feature Envy in the Android apps was relatively poor. The best precision achieved was only 0.25 from JDeodorant and the best recall was only 0.30 from JSpiRIT. Detection of Large Class went significantly better. DECOR was the best performing detector, with decent precision (0.64) and good recall (0.81). For Long Method, DECOR preferred precision above recall, while TACO did the reverse. JDeodorant notably did not return any detections. For Message Chain, DECOR performed poorly. Finally, Spaghetti Code could not be benchmarked as the smell oracle included no actual instances of the smell.

In terms of overlap between the tools, Table 4.3 shows the fraction of detections in the union of two detectors that are in their intersection. In this comparison Message Chain and Spaghetti Code are missing due to the absence of multiple detectors covering these smells. According to the fractions obtained, the detectors have relatively little overlap. The largest overlap occurred with only 0.21 (21%) overlap between DECOR and JspIRIT in the case of the Large Class.

Table 4.3: Overlap between detectors

Detectors	Overlap
Feature Envy	
JDeodorant ∪ JSpIRIT	0.04
JDeodorant ∪ TACO	0.02
JSpIRIT ∪ TACO	0.05
Large Class	
DECOR ∪ JDeodorant	0.13
DECOR ∪ JSpIRIT	0.21
JDeodorant ∪ JSpIRIT	0.18
Long Method	
DECOR ∪ TACO	0.06

4.3 Conclusion

RQ1: How well do existing smell detectors perform on Android apps?

The detectors that were included performed relatively poor on each of the smells. The best result was obtained with DECOR for the Large Class smell (F1 score 0.36), with all other results being significantly worse.

RQ2: Are the smells detected by different detectors the same or are they different?

Considering the fraction of overlapping smell detections, the different tools detect different smell instances. The largest overlap found between detectors was only 0.21, meaning that 79% of smell detections were unique to one of both detectors.

Chapter 5

Study II: Constructing an Android-specific Smell Detector

Using a *machine learning* (ML) approach and selection of code metrics, a novel Android-specific smell detector was constructed in this study. A precedence exists for this approach, performed by Fontana et al. [12] for Java code in general. The main goal of this study was to improve on the performance values as obtained for traditional smell detectors in Chapter 4. The research question for this study was:

- **RQ1** How does a ML-based Android-specific smell detector perform compare to existing Java detectors?

Section 5.1 describes the methodology used to answer the research questions. Section 5.2 presents the results obtained and the corresponding analysis.

5.1 Methodology

The methodology consisted of design choices made in the various stages of the machine learning process.

5.1.1 Machine learning model

Within the field of machine learning, the problem of code smell detection in combination with an oracle fell into the category of *supervised learning*: problems where a dataset of correctly classified objects is available. Indeed, the language constructs added to the smell oracle (Chapter 3) were identified as part of the smelly class, while the language constructs that were not added are implicitly part of the non-smelly class. The union between the oracle and its complement set, also known as the *universe*, is equal to the set of all parsed language constructs that were parsed from the Android source code.

Learning from this dataset of *labeled objects*, supervised learning attempts to classify unlabelled new objects. Within the field of supervised learning, this was an example of a *two-class classification* problem, with smelly and non-smelly as the possible classes.

The Android-specific smell detector was a combination of four distinct two-class classification problems: for the Feature Envy, Large Class, Long Method and Message Chain smell. The Spaghetti Code smell could not be included due to a lack of smelly instances in the smell oracle. With all parsed language constructs labelled as non-smelly, an ML-approach would have no data to learn about the smelly class. The traditional supervised learning model as applied to smell detection is shown in Figure 5.1.

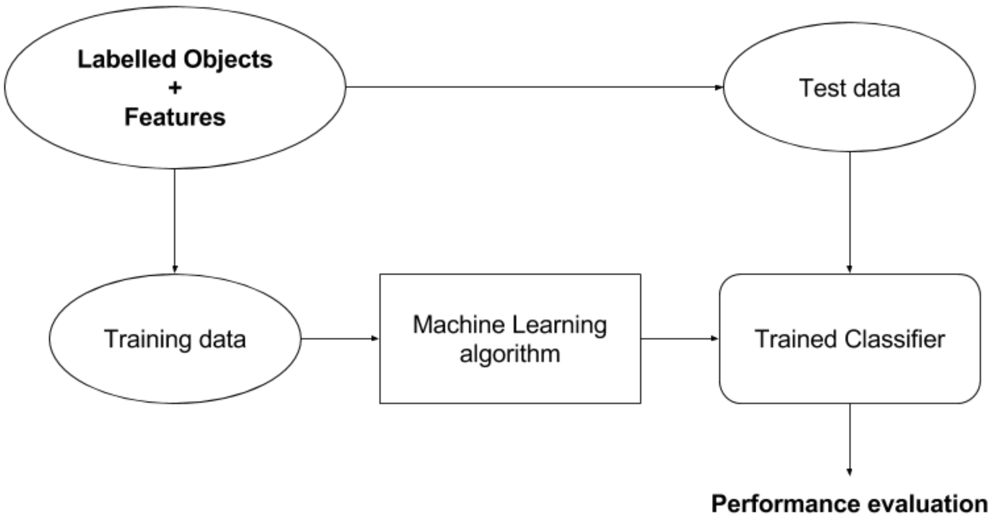

Figure 5.1: Supervised learning model.

This model was applied four times for the different code smells. The smell oracle and set of all parsed language constructs were used to derive a labelled dataset. For the *features* of the objects, a combination of classification outputs from the existing detectors investigated in Study I and code metrics chosen using the analysis from the manual smell detection process in Section 3.3 was used.

The set of labelled objects was split into a training and test set to prevent *overfitting*. Overfitting would result in overly optimistic performance results, occurring whenever training- and test data overlap. Many ML-algorithms exist, based on different probabilistic models. A limited selection of popular algorithms was chosen and compared in terms of detection performance. No single best algorithm exists for all problems, therefore the construction of the Android-specific detector was an exercise of exploration.

5.1.2 Design Choices

Weka [27] was used to design the classifier. Weka is a suite of machine learning software which provides data visualization tools and implementations of machine learning algorithms. Multiple design choices were made within Weka to obtain the optimal classifiers.

Class balancing The two classes in the labelled dataset were strongly *imbalanced*. Due to the nature of code smells, the ratio between smelly and non-smelly objects was heavily skewed towards the non-smelly class. This posed a problem, as ML-algorithms would simply 'learn' to classify every unknown new object as non-smelly and achieve great performance. The trained classifier would however learn nothing about distinguishing between the two classes, through the supplied features. Within Weka, a class balancer filter was applied to compensate this. This filter attached weights to the objects in the dataset to effectively give the smelly/non-smelly classes a 50-50 distribution.

Feature normalization Some ML-algorithms do not work properly without feature normalization. For example, when using the Euclidean distance between objects in feature space, features with a larger spread in values will more strongly contribute to this distance. Within Weka, a normalization filter was applied. It scaled all numeric features to the [0,1] range.

Chosen Features The features used were the classification outputs of existing detectors combined with manually determined code metrics based on the analyses made in Section 3.3. Broken down by code smell, the chosen code metrics were:
Feature Envy (method objects)

- `internalEnvy`: Numeric, the value of envy (Eq. 3.1) between the method and its parent class.
- `externalEnvy`: Numeric, the largest envy value between the method and an external class.
- `numberOfEnviedClasses`: Numeric, the number of external classes with higher envy compared to internal envy.
- `parentClassInheritsFromAndroidClass`: Nominal, a boolean string stating whether the type hierarchy of the parent class contains a qualified name of the pattern `android.*`.
- `jdeodorant, jspirit, taco`: Nominal, boolean strings stating whether or not each respective smell detector classified this method as smelly.

Large Class (class objects)

- `loc, noa, nom`: Numeric, the lines of code, number of attributes and number of methods metric values.
- `locSumOfInnerClasses`: Numeric, the lines of code of the class subtracted by the sum of lines of code of any inner classes.
- `numberOfAnnotatedOverride`: Numeric, the number of methods in the class explicitly marked using `@Override`.
- `parentClassInheritsFromAndroidClass`: The same feature as was added for the previous smell.

- `decor, jdeodorant, jspirit`: Nominal, boolean strings stating whether or not each respective smell detector classified this class as smelly.

Long Method (method objects)

- `locOfLargestCodeBlock`: Numeric, the lines of code of the largest code block in the method. In Java, each *block* is a section of code surrounded by accolades. This includes if-statements and for-loops.

- `parentClassInheritsFromAndroidClass`: The same feature as was added for the previous smell.

- `decor, jdeodorant, taco`: Nominal, boolean strings stating whether or not each respective smell detector classified this class as smelly.

Message Chain (invocation chain objects)

- `numberOfTraversedClasses`: Numeric, number of distinct classes traversed in the invocation chain.

- `numberOfTraversedMutableClasses`: Numeric, similar to the previous feature but excludes classes of the patterns `android.*` and `java.*`, which can not be refactored.

- `parentClassInheritsFromAndroidClass`: The same feature as was added for the previous smell.

- `decor`: Nominal, boolean string stating whether or not decor classified this invocation chain as smelly.

Explored Classifiers With no single classifier that works best for all classification problems, a selection of mainstream classifiers was explored. The classifiers were intentionally chosen to use different underlying mathematical approaches. Many of the classifiers contain parameters which could be tweaked. Within the scope of this study these were left at their default Weka values. In consultation, the following classifiers were explored:

- `functions.SMO` Support vector classifier.

- `trees.REPTree` Decision tree learner.

- `trees.RandomForest` Constructs a forest of random trees.

- `functions.Logistic` Logistic classifier.

- `weka.classifiers.bayes.NaiveBayes` Application of *Bayes' rule* and the assumption that all features are independent.

- `weka.classifiers.lazy.IBk` A k-nearest neighbour classifier.

Performance Evaluation Trained classifiers were evaluated using 10-fold cross validation. With respect to class balancing, balancing was applied to the training data only for each fold.

5.2 Results

The obtained evaluation results are shown in Table 5.1. As a performance *baseline*, the best performing smell detector in terms of F1 score from Study I was included as well. By default, Weka's evaluation results are a weighted average over all classification classes. However, the labelled dataset was heavily skewed towards the non-smelly class while the class of interest is the smelly class. Therefore, the metrics as shown are calculated over the smelly classification class only.

Table 5.1: Classifier performance, for 'smelly' classification class only.

Detector	TP	FP	FN	Precision	Recall	F1 score
Feature Envy						
JDeodorant (baseline)	10	30	74	0.25	0.12	0.08
SMO	84.0	880.0	1.0	0.09	0.99	0.16
REPTree	78.0	802.0	7.0	0.09	0.92	0.16
RandomForest	46.0	416.0	39.0	0.10	0.54	0.17
NaiveBayes	82.0	823.0	3.0	0.09	0.96	0.17
Logistic	83.0	850.0	2.0	0.09	0.98	0.16
IBk	48.0	410.0	37.0	0.10	0.56	0.18
Large Class						
DECOR (baseline)	35	20	8	0.64	0.81	0.36
SMO	41.0	74.0	2.0	0.36	0.95	0.52
REPTree	41.0	22.0	2.0	0.65	0.95	0.77
RandomForest	40.0	15.0	3.0	0.73	0.93	0.82
NaiveBayes	42.0	71.0	1.0	0.37	0.98	0.54
Logistic	41.0	36.0	2.0	0.53	0.95	0.68
IBk	22.0	18.0	21.0	0.55	0.51	0.53
Long Method						
DECOR (baseline)	18	11	40	0.62	0.31	0.21
SMO	56.0	481.0	4.0	0.10	0.93	0.19
REPTree	49.0	281.0	11.0	0.15	0.82	0.25
RandomForest	35.0	235.0	25.0	0.13	0.58	0.21
NaiveBayes	57.0	520.0	3.0	0.10	0.95	0.18
Logistic	56.0	535.0	4.0	0.09	0.93	0.17
IBk	28.0	228.0	32.0	0.11	0.47	0.18
Message Chain						
DECOR (baseline)	9	94	92	0.09	0.09	0.04
SMO	101.0	375.0	0.0	0.21	1.00	0.35

Table 5.1: Classifier performance, for ‘smelly’ classification class only.

Detector	TP	FP	FN	Precision	Recall	F1 score
REPTree	101.0	395.0	0.0	0.20	1.00	0.34
RandomForest	99.0	386.0	2.0	0.20	0.98	0.34
NaiveBayes	101.0	483.0	0.0	0.17	1.00	0.29
Logistic	94.0	1132.0	7.0	0.08	0.93	0.14
IBk	98.0	385.0	3.0	0.20	0.97	0.34

For this study, classifier performance was compared on the F1 score only, with precision and recall as informative metrics. While some classifier designs may prefer better precision at the cost of recall, or the reverse, a neutral stance was held here. Therefore, the F1 score (harmonic mean between precision and recall) was a usable single metric of performance.

Overall, the various classifiers perform better compared to a single smell detection tool. This was not unexpected, as the available predictors can include the output of multiple detectors. The code metrics that were added were also derived from the analysis used to create the smell oracle. As the exception, the Long Method smell was not detected much better compared to the baseline.

From this table, the optimal Android-specific smell detector would contain a trained classifier corresponding to algorithm with the best F1 score as seen in the table. For reference, an Eclipse project containing all machine learning data and an implementation of classifier evaluation is available on GitHub[1].

5.3 Conclusion

RQ1: How does a ML-based Android-specific smell detector perform compared to existing Java detectors?

The ML-based Android-specific smell detector performed better than the baseline for the Feature Envy, Large Class and Message Chains smells, while performing similar for the Long Method smell. For Feature Envy, the IBk k-nearest neighbour classifier performed best with a F1 score of 0.18 (baseline 0.08). For Large Class, the RandomForest tree classifier performed best with a F1 score of 0.82 (baseline 0.36). For Long Method, the REPTree tree classifier performed best with a F1 score of 0.25 (baseline 0.21). Finally, for Message Chain, the SMO support vector classifier performed best with a F1 score of 0.35 (baseline 0.04).

Possible paths to improve the ML-based detector include the adding better code metrics where possible, performing feature selection to discard weak predictors, exploring more types of classifier algorithms and tweaking the parameters of the classifier algorithms that were used.

[1]`https://github.com/DustinLim/msc-thesis-weka-classifier-evaluation`

Chapter 6

Threats to Validity

This chapter discusses possible threats to validity.

Interpretation of Code Smells Some of the definitions of the investigated code smells contained abstract elements. As a result, their interpretation is to a certain extent subjective. We attempted to clearly point out any abstract elements in terms of definitions in Section 2.2, followed by our interpretation during the construction of preprocessor filtering rules and the manual validation of code smells. The consultation of previous work attempted to reduce the risk of misinterpreting the investigated code smells. A common mitigation is the use of multiple researchers which are required to reach consensus.

Preprocessor Filtering Rules The usage of filtering rules in the preprocessor introduced an internal threat to validity. If the filtering rules were to remove any smelly instances, the validity of the smell oracle would be affected. The utilization of filtering was required due to the infeasibility of manually inspecting the complete dataset of language constructs. For each code smell, we supplied a justification for why filtered out instances were very unlikely to be smelly.

Generalization of Results Threats to external validity include the limited sample size of Android apps included in the smell oracle, used to determine the detection performance of various techniques and tools. In Study I, a limited set of Java smell detectors was included in the determination of detection performance from existing smell detectors. In Study II, the smell oracle was a key component for the machine learning approach used. The limited scope of the smell oracle could affect the generalizability of the Android-specific smell detector that was constructed. All of these cases were a based on a consideration between generalization of results and limited time resources. An attempt was made to include as much data as possible.

Chapter 7

Conclusion

This chapter gives an overview of the thesis's contributions and conclusions. This is followed by some ideas for future work.

Contributions In terms of software artefacts, the source code preprocessor as described in Chapter 3.2 is publicly available on GitHub [1]. Functionality includes the parsing of language constructs from Android code and the application of filtering rules. The preprocessor generates an output file containing all preprocessed entities, including uniquely generated identifiers.

A second software artefact is the Weka classifier evaluator, also publicly available on GitHub[2]. This tool accepts a list of unique identifiers which representing a smell oracle and lists of smell detections from existing detectors. The evaluator generates a feature space from this input, followed by the training and evaluation of Weka classifiers.

A final contribution is the manually constructed smell oracle for the five investigated Android apps. This oracle is included as one of the input files for the Weka classifier evaluator.

Conclusions The construction of the smell oracle was very time consuming. In the context of Android code, a pre-existing dataset of identified code smells did not exist, necessitating the manual construction of a smell oracle. Given the amount of manual work involved, the added value of the existence of an effective code smell detector becomes very apparent. The completed smell oracle contained positive examples for four of the five code smells investigated. The Spaghetti Code smell was left with zero positive examples.

In Study I, we concluded that existing smell detectors perform relatively poorly on Android apps. However, the instances that were found using existing detectors were shown to overlap very little. This indicated that the combination of multiple detectors could improve smell detection performance.

In Study II, we concluded that a novel machine learning approach designed specifically for Android code performed significantly better in three of the investigated code smells.

[1]`https://github.com/DustinLim/mscthesis-preprocessor`

[2]`https://github.com/DustinLim/msc-thesis-weka-classifier-evaluation`

Classifiers were trained on a combination of the output of existing smell detectors and selected code metrics. Different Weka classifiers turned out to work best for the different code smells.

Future work Many opportunities are left for future work. The Android-specific smell detector can be improved through the addition of more Android apps to the smell detector. In addition, the feature space of the machine learning process can be expanded by added more existing detectors or code metrics. Finally, different classifiers may be evaluated and tweaked in terms of parameters to improve on the performance results that were obtained.

Bibliography

[1] Android ndk documentation. https://developer.android.com/ndk/guides/index.html.

[2] Android package index. https://developer.android.com/reference/classes.html.

[3] Android studio. https://developer.android.com/studio/index.html.

[4] Darren C Atkinson and Todd King. Lightweight detection of program refactorings. In *Software Engineering Conference, 2005. APSEC'05. 12th Asia-Pacific*, pages 8–pp. IEEE, 2005.

[5] Gabriele Bavota, Rocco Oliveto, Malcom Gethers, Denys Poshyvanyk, and Andrea De Lucia. Methodbook: Recommending move method refactorings via relational topic models. *IEEE Transactions on Software Engineering*, 40(7):671–694, 2014.

[6] William H Brown, Raphael C Malveau, Hays W McCormick, and Thomas J Mowbray. *AntiPatterns: refactoring software, architectures, and projects in crisis*. John Wiley & Sons, Inc., 1998.

[7] Checkstyle. http://checkstyle.sourceforge.net.

[8] Eclipse ide. https://www.eclipse.org/home/index.php.

[9] F-droid - free and open source android app repository. https://f-droid.org.

[10] Marios Fokaefs, Nikolaos Tsantalis, Eleni Stroulia, and Alexander Chatzigeorgiou. Identification and application of extract class refactorings in object-oriented systems. *Journal of Systems and Software*, 85(10):2241–2260, 2012.

[11] Francesca Arcelli Fontana, Pietro Braione, and Marco Zanoni. Automatic detection of bad smells in code: An experimental assessment. *Journal of Object Technology*, 11(2):5–1, 2012.

[12] Francesca Arcelli Fontana, Marco Zanoni, Alessandro Marino, and Mika V Mantyla. Code smell detection: Towards a machine learning-based approach. In *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, pages 396–399. IEEE, 2013.

[13] Martin Fowler and Kent Beck. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

[14] gradle-eclipse-aar-plugin. `https://github.com/ksoichiro/gradle-eclipse-aar-plugin`.

[15] Michele Lanza and Radu Marinescu. *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems*. Springer Science & Business Media, 2007.

[16] Robert C Martin. *Agile software development: principles, patterns, and practices*. Prentice Hall, 2002.

[17] Naouel Moha, Yann-Gael Gueheneuc, Laurence Duchien, and Anne-Francoise Le Meur. Decor: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering*, 36(1):20–36, 2010.

[18] Emerson Murphy-Hill and Andrew P Black. An interactive ambient visualization for code smells. In *Proceedings of the 5th international symposium on Software visualization*, pages 5–14. ACM, 2010.

[19] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Fausto Fasano, Rocco Oliveto, and Andrea De Lucia. On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. *Empirical Software Engineering*, pages 1–34, 2017.

[20] Fabio Palomba, Annibale Panichella, Andrea De Lucia, Rocco Oliveto, and Andy Zaidman. A textual-based technique for smell detection. In *Program Comprehension (ICPC), 2016 IEEE 24th International Conference on*, pages 1–10. IEEE, 2016.

[21] Fabio Palomba, Pasquale Salza, Adelina Ciurumelea, Sebastiano Panichella, Harald Gall, Filomena Ferrucci, and Andrea De Lucia. Recommending and localizing change requests for mobile apps based on user reviews. In *Proceedings of the 39th International Conference on Software Engineering*, pages 106–117. IEEE Press, 2017.

[22] Pmd source code analyzer. `https://pmd.github.io`.

[23] Sourcemaking - code smells. `https://sourcemaking.com/refactoring/smells`.

[24] Nikolaos Tsantalis and Alexander Chatzigeorgiou. Identification of move method refactoring opportunities. *IEEE Transactions on Software Engineering*, 35(3):347–367, 2009.

[25] Nikolaos Tsantalis and Alexander Chatzigeorgiou. Identification of extract method refactoring opportunities for the decomposition of methods. *Journal of Systems and Software*, 84(10):1757–1782, 2011.

[26] Santiago Vidal, Hernan Vazquez, J Andres Diaz-Pace, Claudia Marcos, Alessandro Garcia, and Willian Oizumi. Jspirit: a flexible tool for the analysis of code smells. In *Chilean Computer Science Society (SCCC), 2015 34th International Conference of the*, pages 1–6. IEEE, 2015.

[27] Weka 3: Data mining software in java. `https://www.cs.waikato.ac.nz/ml/weka/`.

[28] Apple xcode xed documentation. `https://developer.apple.com/legacy/library/documentation/Darwin/Reference/ManPages/man1/xed.1.html`.

Appendix A

Manual Smell Detection Analysis

A.1 Feature Envy

Analysis	Instances
[Rejected] Method delegates its invocation to another class.	94
[Accepted] Refactoring (part of) method appears appropriate.	82
[Rejected] Envy is evenly spread over multiple classes.	76
[Rejected] Method uses an utility class.	59
[Rejected] Envy is towards superclass of the class declaring the method.	58
[Rejected] Method manages collection of envied class.	56
[Rejected] No obvious refactoring possible.	49
[Rejected] Envied fields are public constants.	37
[Rejected] Envy not applicable towards logging class.	34
[Rejected] Envied class is instantiated, followed by setters.	34
[Rejected] Method is too large to analyse/refactor.	33
[Rejected] Parent class handles UI, envied class is part of the Model.	24
[Rejected] Method is inside a (static) utility class.	21
[Rejected] Parent class handles UI, envied class is child UI element.	13
[Rejected] Method overrides superclass implementation.	12
[Rejected] Parent class handles UI, envied class is parent Activity.	11
[Rejected] Method is inside a utility-type class.	8
[Rejected] Envied class is an interface.	7
[Rejected] Envied class is an anonymous inline class.	6
[Rejected] Method is inside anonymous class declaration.	5
[Rejected] Parent class is designed as layer above envied class.	4
[Rejected] Method accesses static fields / methods of envied class.	2
[Rejected] False positive due to logging code.	2
[Rejected] Static utility method, should be moved to envied utility class.	1
[Rejected] Parent Android View class, move method to envied Activity.	1

[Rejected] Parent Android Activity class, managing the envied View.	1
Total accepted instances:	84
Total rejected instances:	646

A.2 Large Class

Analysis	Instances
[Rejected] Class size inflated due to inner class(es).	16
[Accepted] Multiple class responsibilities visible.	16
[Rejected] No obvious refactoring.	12
[Accepted] Too many attributes.	10
[Accepted] UI class manages too much UI.	10
[Accepted] Too many methods.	5
[Accepted] Multiple methods with: @SuppressWarnings(”unused”)	2
[Accepted] Part of class should be moved to a Model class.	2
[Rejected] Class structure (attributes and methods) is clear.	1
[Accepted] Implements too many interfaces.	1
[Accepted] Repeated switch-statements imply extractable subclasses.	1
[Rejected] Contains SDK method overrides and UI math attributes.	1
[Rejected] Class inflated due to Large Method(s).	1
Total accepted instances:	43
Total rejected instances:	31

A.3 Long Method

Analysis	Instances
[Accepted] Visible method section(s) to extract.	18
[Accepted] Contains large inline anonymous class(es).	17
[Rejected] Contains if-else or switch construct with many small cases.	15
[Accepted] Contains if-else conditional with large cases.	14
[Rejected] Method performs single focused task.	9
[Accepted] Builds a UI element. Larger method sections for sub-elements.	5
[Rejected] A long sequence of short if-statements.	4
[Accepted] Complex, deeply nested control flow.	2
[Accepted] Contains switch construct with large cases.	1
[Accepted] Manually draws many UI elements, grouping possible.	1

[Rejected] Builds a UI element. Many small method sections for sub-elements.	1
Total accepted instances:	58
Total rejected instances:	29

A.4 Message Chain

Analysis	Instances
[Rejected] Some invoked types belong to the Java platform.	139
[Rejected] Some invoked types belong to the Android platform.	91
[Accepted] Hide Delegate applicable.	84
[Rejected] Part of chain invokes inherited method(s).	57
[Rejected] Static method pattern is present.	50
[Rejected] Part of chain invokes local method(s).	38
[Rejected] Chain invokes a class and a Java collection of the same.	30
[Rejected] Utility-type interface is used.	30
[Accepted] Refactoring is appropriate.	17
[Rejected] Invocation on an inherited method.	12
[Rejected] Chain structure is justified.	12
[Rejected] Chained calls that repeatedly return the object itself.	6
Total accepted instances:	101
Total rejected instances:	465