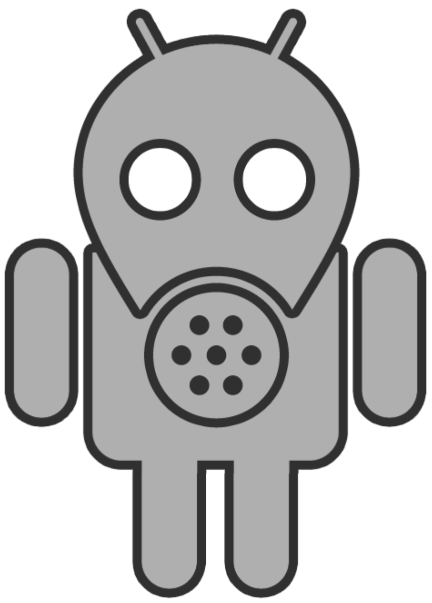
Code Smells in the Mobile Applications Domain

Master Thesis



Daniël Verloop

Code Smells in the Mobile Applications Domain

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Daniël Verloop
born in Vlaardingen, the Netherlands

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

Code Smells in the Mobile Applications Domain

Author:	Daniël Verloop
Student id:	1188526
Email:	`d.verloop@student.tudelft.nl`

Abstract

The mobile applications market is growing rapidly, over 85 billion mobile applications have been downloaded. Smartphone sales are already bigger than computer sales and this might become the first year in which over one billion smart phones will be sold. Regardless of these statistics there is not a lot of research to be found on the subject.

In this thesis one of the observations done in a recent study on mobile applications is reproduced. We also look for code smells (patterns in source code that are associated with bad design and bad programming practices) in a number of commercial and open source applications. The results of this analysis is used to determine if certain code smells have a higher likelihood to appear in mobile application source code.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Mobile Applications

Mobile Applications are software applications that run on mobile systems like smartphones or tablets. Applications are installed on a mobile system using an application store which downloads and installs applications chosen by the user. Both the number of applications available in these stores and their download counts are growing rapidly as can be seen from Table 1.1.

	Android		**iOS**	
Approximate Year/Month	**Applications Available**	**Accumulated Downloads**	**Applications Available**	**Downloads to date**
2009/03	2300		25000	0.8 billion
2010/08	80000	1 billion	250000	6.5 billion
2011/05	200000	3 billion	425000	14 billion
2012/06	600000	20 billion	650000	30 billion
2013/04	850000	40 billion	825000	45 billion

Table 1.1: Statistics of the two major application stores. Source: Wikipedia

Mobile applications run on the software layer of the mobile system which is called the mobile software platform. Currently the most common mobile software platforms are Android, iOS, Windows Phone and Blackberry. Table 1.2 shows the worldwide smartphone sales for the first quarter of 2012 and 2013 based on operating system. Given the rise in sales it is to be expected that this year for the first time over one billion smartphones will be sold.

As can be seen from both tables above the market for mobile applications and the sales of devices on which these applications run is still growing.

Operating System	**2013 Q1**		**2012 Q1**	
	Units	**%**	**Units**	**%**
Android	156,186	74.4 %	83,684	56.9 %
iOS	38,332	18.2 %	33,121	22.5 %
Blackberry	6,219	3.0 %	9,940	6.8 %
Windows Phone	5,990	2.9 %	2,723	1.9 %
Others	3,321	1.5 %	17,554	11.9 %
Total	**210,048**	**100.0**	**147,022**	**100.0**

Table 1.2: Worldwide smartphone sales to end users by Operating System (in thousands of units). Source: Gartner (May 2013)

1.2 Code Smells

Code smells are patterns in source code that are associated with bad design and bad programming practices. Unlike programming errors they do not result in incorrect external behavior. Code smells point to areas in an application that could benefit from refactoring. Refactoring is defined as *'a technique for restructuring an existing body of code, altering its internal structure without changing its external behavior"* [11].

Choosing not to resolve code smells by refactoring will not result in the application failing to work but will likely increase the difficulty of maintaining it. Thus refactoring helps to improve the maintainability of an application. Given that maintenance is considered the most expensive stage of software development and that the proportion of maintenance cost over total software cost is increasing each decade [18] refactoring will save money in the long haul.

Because code smells are described in terms of program patterns they can be identified using static analysis (code analysis), opposed to behavior patterns where dynamic (runtime) information is needed. This means a tool searching for code smells does not need to run the application. It only needs access to the source code and optionally resources and libraries that are referenced by the source code. Using tools to find code smells and refactor the code will help improve the maintainability of applications.

1.3 Problem Statement

The market for mobile applications is growing fast:

- The worldwide smartphone sales for the first quarter has increased by 25% in a single year (see Table 1.2).
- The number of mobile applications available from the various application stores and their download count is increasing rapidly (see Table 1.1).

At the same time the software development process is changing at a pace unknown to traditional software systems: for most of the mobile systems a major update is released at least once a year. The combination of fast growth and rapid updates implies that it is important for mobile software developers to keep their applications maintainable. If mobile software developers fail or are unable to keep their application maintainable it will be more difficult for them to keep up with the rapid changes and ensure their application remain relevant in the fast changing mobile application market.

The speed with which the mobile application market is changing is not the only difference between mobile software applications and traditional software applications. According to [15] mobile applications are substantially different from traditional software applications. Some of these differences are obvious like the lack of a permanent power supply which requires the developer to be energy efficient. Others are less obvious like a short lifespan of mobile applications or that the use of inheritance is almost entirely absent in mobile applications which increases the difficulty of maintaining it. Because of these differences, research done on traditional software applications may not apply to mobile applications and despite the growth and changes of the mobile market there is not a lot of research to be found on this new area of software development.

Given the new challenges developers face while developing mobile applications and the limited amount of research on the subject it will be interesting to research how the new mobile software developers can limit the number of code smells in their applications and thereby increase the maintainability of these applications.

1.4 Research Questions

Given the problem statement from the previous section, the following three research questions will be addressed:

- **RQ1**: What are the main differences between mobile software applications and traditional software applications?
- **RQ2**: Which tools can be used to find code smells in mobile applications and what is the quality of these tools?
- **RQ3**: Is mobile application code more prone to code smells and if it is how can this risk be limited?

1.5 Thesis Structure

This thesis is composed of the following seven chapters:

- **Chapter 1**: Introduction.
- **Chapter 2**: Background into mobile systems and code smells.
- **Chapter 3**: Related work on the subject.
- **Chapter 4**: The framework of the research.
- **Chapter 5**: The analysis results.
- **Chapter 6**: Recommendations gathered from the results of the research performed.
- **Chapter 7**: Conclusions and future work.

Chapter 2

Background

In this chapter background information is given about mobile operating systems and Android in particular, to comprehend the environment a developer has to work with and his application operates in.

2.1 Mobile Systems

2.1.1 Software Platforms

As said in the introduction there are currently four major mobile software platforms: Android, iOS , Blackberry and Windows Phone.

Android

Android is an open source software stack for mobile devices that includes an operating system (based on the Linux kernel), middleware (application framework, libraries and runtime) and a number of applications (image viewer, web browser, etc). It can be found on smartphones, tablets, netbooks, media players, radios, televisions and many more electronic devices. Android is developed by Google and the Open Handset Alliance. It is currently the operating system with the most momentum and the only one increasing its world-wide sales to end users between the last quarter of 2012 and the first quarter of 2013.

iOS

iOS (previously iPhone OS) is a mobile operating system developed and distributed by Apple Inc. Originally released in 2007 for the iPhone and iPod Touch, it has been extended to support other Apple devices such as the iPad and Apple TV. Apple does not allow iOS installations on non-Apple hardware unlike Android and Windows Phone. In sales numbers iOS is currently a distant second to Android but it was the first in building a large ecosystem around mobile applications so iOS has a distinct advantage over the other players. It took Android until early 2013 to surpass iOS in absolute numbers of available applications in their respective application stores.

Windows Phone

Windows Phone is a proprietary mobile operating system developed by Microsoft. It is the successor of Windows Mobile. Currently Windows Phone does not appear to gain any momentum given the fact that world-wide sales to end users dropped between the last quarter of 2012 and the first quarter of 2013.

Blackberry 10

Blackberry 10 is a proprietary mobile operating system developed by BlackBerry Limited (formerly Research In Motion, RIM). It is the successor of Blackberry OS which until a few years ago was the most used mobile software platform. The sales figures of the Blackberry line of smartphones has been dropping since early 2011.

2.1.2 Problem Domain

The research in this thesis will be on the Android mobile software platform for a number of reasons:

- Android currently is the most populair mobile software platform: almost 75% of all smartphones sold in the previous quarter have Android installed.
- Android applications are mostly written in Java. Since Java is the preferred language for academic use it will make finding analyzing tools easier.
- The author's familiarity with the Android OS, Java and Eclipse (the default Android IDE).

2.2 Android System

Android applications run on top of the Android system, Figure 2.1 shows where applications are located in the Android system architecture.

- The top layer, **applications**, is where Android applications are located. They interact directly with the blue parts: other applications, the application framework and the core libraries.
- The **core libraries** in the Android runtime consists of Java class libraries comparable to the Java SE libraries but are based on a subset of the Apache Harmony Java implementation.
- The applications run inside the **Dalvik virtual machine**. It runs Dalvik executable files.

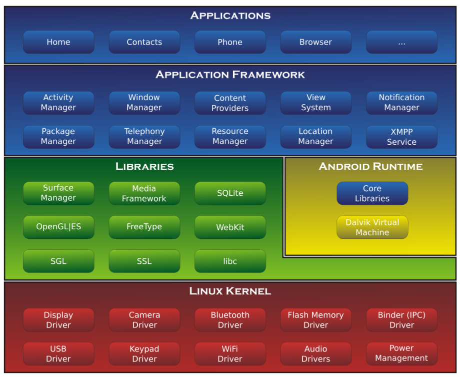


Figure 2.1: The Android system architecture.

2.2.1 Android Applications

Overview

An Android application consists of a single APK file containing the compiled source code, resources, data files and the manifest.

Application Components

Android applications are written in Java, extended with some native parts in C and screen layouts in XML. Android applications are composed of one or more application components: activities, services, content providers and broadcast receivers.

- An **Activity** is an Android component that provides a single screen with a user interface that users can interact with, e.g. sending an email or taking a picture. The first Activity the user sees is called the Main Activity.
- A **Service** is an Android component that performs a long-running operation in the background, e.g. in a music application there is a service that plays an audio file.
- **Content providers** manage access to structured sets of data. They are the standard interfaces to connect data in one process with code running in another process, e.g. a content provider that gives an application access to a list of all the contacts in the phone.
- A **Broadcast receiver** is the base class to receive messages sent by sendBroadcast(), e.g. a message is sent by the Android system when the internet connection is lost.

Three of these components are started using messages which in Android are called Intents. Intents are defined by the Android API as

> *"Three of the core components of an application (activities, services, and broadcast receivers) are activated through messages, called Intents. Intent messaging is a facility for late run-time binding between components in the same or different applications. The intent itself, an Intent object, is a passive data structure holding an abstract description of an operation to be performed or in the case of broadcasts a description of something that has happened and is being announced."* [1]

The user interface of an Android application is defined by a layout that is declared inside a XML file or in code. Each user interface component in Android extends the *android.widget.View* class.

Other Android components

Besides the application components there are other important components:

- An **Application** instance is created when an application process is first started. The main use of it is to maintain a global application state.
- **Fragments** are used to split up the behavior of an Activity which helps developing for different screen sizes. For example, the gmail application has 2 fragments: the email list and the email content; on phones the fragments are shown in two separate screens, on tablets both fragments are shown in one screen.
- **Adapters** act as a bridge between an AdapterView (e.g. a list or a grid) and the data shown in the view. The Adapter provides access to the data, mostly in the form of the getCount and getItem methods. The Adapter is also used to create the view and set the contents of the view.

Resources

The res directory contains all resources (e.g. images, translated strings, layouts, colors, sizes, styles, etc..). To reference these resources in source code Android generates an R class file where each type of resource has its own subclass and each resource file has its own resource id in the subclass. For example R.layout.mainscreen will reference the layout XML file located at layout/mainscreen.xml.

Android Manifest

The Android manifest specifies the components in the application, declares the application requirements and some other application properties. Here is an example of the application properties:

- Minimum required API version
- Target API version
- Hardware requirements e.g. camera, touch screen, GPS
- Permissions given by the user e.g. access to contact information
- Libraries required e.g. Google Maps
- Screen densities
- Minimum OpenGL version

A developer has to specify in the manifest which Activity is the main activity, i.e. the entry point to the application. The developer also has to specify whether that main activity should appear in the list of activities (called the launcher) where it can be launched from. If an application has only one screen than that Activity will be the main activity and will appear in the launcher.

Processes and Threads

Each Android application and all its components run in a separate process. There are five priority levels an Android process can have:

- Foreground process: The process that is required for what the user is currently doing. A process is in the foreground if it hosts any of the following applications components:
  - An Activity the user is interacting with.
  - A Service bound to an Activity the user is interacting with.
  - A Service with a high priority.
  - A Service running one of its lifecycle methods.
  - A BroadcastReceiver running its onReceive method.
- Visible process: A process that is not in the foreground but can still affect what the user sees on the screen.
- Service process: A process that is running a service which does not fall in the previous categories, e.g. a service playing music or downloading a large file.
- Background process: A process with an Activity that is not visible to the user. These activities are kept in a background process so they can quickly go to the foreground.
- Empty process: A process that does not hold any of the four application components. This process is only kept alive for caching purposes, to improve the startup time when it is needed.

Saving and restoring the application state is done with two methods, the onSaveInstanceState() method which is called when the Activity leaves the foreground and the onRestoreInstanceState() method which is called when the Activity returns to the foreground.

Activity Lifecycle

The Activity lifecycle describes which methods are called when the state of the Activity changes. These methods are called the lifecycle methods. Figure 2.2 shows the state transitions and the corresponding lifecycle methods.

The lifecycle methods that are most used are:

onCreate() The first method called when an activity is launched. It is used to set up the layout of the Activity. This includes setting up the user interface using *setContentView*, getting all references to the UI components using *findViewById()*, setting the UI component listeners and setting all window properties like fullscreen and theme.

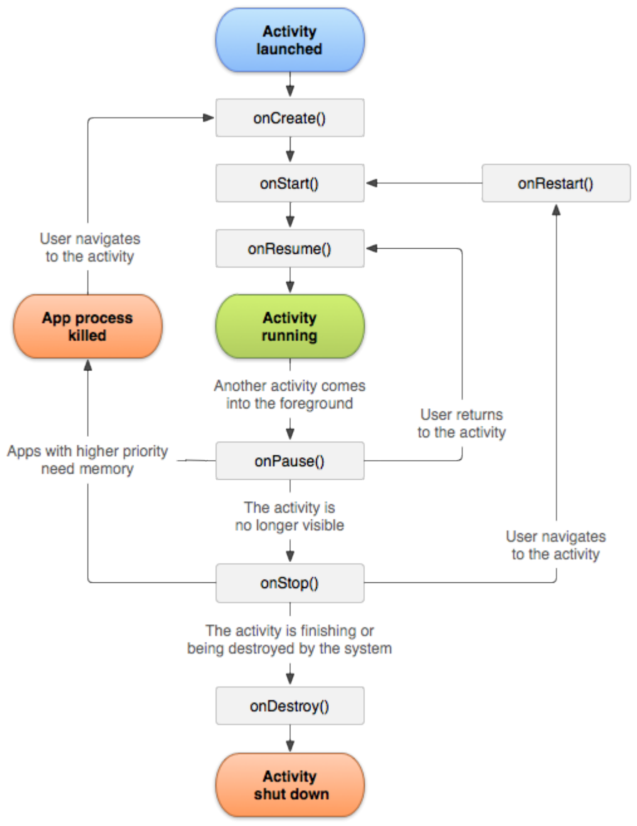

Activity launched
onCreate()
onStart()
onRestart()
User navigates to the activity
onResume()
App process killed
Activity running
Another activity comes into the foreground
User returns to the activity
Apps with higher priority need memory
onPause()
The activity is no longer visible
User navigates to the activity
onStop()
The activity is finishing or being destroyed by the system
onDestroy()
Activity shut down

Figure 2.2: The Android Activity life cycle by state paths.

onResume() Just before the Activity comes to the foreground onResume() is called. It is used to set the content of the user interface and to register any Android system listeners (like a gps location listener).

onPause() When the Activity is no longer shown in the foreground onPause() is called. It is used to save any content in the user interface (like the current search text).

onDestroy() When the activity is finished or destroyed by the system onDestroy() is called. It is used to perform the final clean up of the Activity.

Software Development Kit

The Android SDK consists of all the necessary tools to build Android applications. It also contains an Eclipse plugin called Android Development Tools (ADT) which integrates into the Eclipse IDE.

It includes the Android SDK Manager to download and update Android packages. The package list consists of tools, Android platform API's (documentation, SDK library, samples, emulator images and sources) and optional libraries like the Android Support Library (backward compatibility library), Admob Ads SDK (for including advertisement in your application), etc..

2.2.2 Development

Below is a list of the steps performed while developing an Android application to give an idea of the development process.

Setting up the project

The development of an Android application begins with the creation of an the Android project consisting of:

- Application properties e.g. application name, project name, package name, minimum API, target API and user interface theme.
- Template for the main activity:
  - BlankActivity which is almost empty.
  - FullscreenActivity which functions to show/hide the system UI (status bar, navigation / system bar) and the action bar.
  - LoginActivity which has a textfield for an email address, a textfield for a password and a button to start the login process.
  - MasterDetailFlow which uses fragments to show two columns on large screens like tablets and one column on smaller screens like phones.

Implementation

- The functionality is implemented in the main activity. Any third-party libraries to be used are added into the library directory.
- Images to show in the application are added to the drawable directories.
- Components are added to the layout using a graphical layout editor with drag and drop support or using a XML text editor
- Labels, titles, error messages or any other text are added to the application by the the strings.xml resource file. This makes it easy to translate the application into other natural languages because the Android system chooses the right strings.xml to get the text from depending on the current system locale.

Build Process

Android applications are build by making an Android application package file (APK file). This process consists of three steps that are described below:

- The first step is the pre-compilation step which consist of generating Java source files. For example, a *R.java* source file is generated which references the resources in the res directory.

- The next step is the compilation step where the Java code is first compiled into Java bytecode followed by converting this Java bytecode into Dalvik bytecode because Android applications run on the Dalvik VM.
- The third step consist of creating the APK file by bundling the Dalvik executable, resources and assets. After creating the APK it will be signed by either a debug or release key depending on the type of build process.

The result of this final step is a single APK file which contains a complete Android application to be installed on Android devices or uploaded to Android application markets like the Google Play Store or the Amazon Appstore.

Testing

Testing in Android is done with Android JUnit test cases, Android uses an extended JUnit library for testing of Android components. Because Android components have a specific life cycle there is a set of control methods into the Android system called Android instrumentation for testing of activities, content providers and services.

Maintenance

If the developer wants the app to remain relevant he will have to keep the application up to date with the latest Android platform API. Every new major version of Android has had a significant user interface change that required the developer to make some (minor) changes to the application to prevent it from looking out of place.

2.2.3 Unique Properties

Mobile application systems have some unique properties which affect the development process. This section lists the most important properties that makes developing for mobile applications different from developing for traditional software applications.

Small Number of Developers

Mobile applications are usually developed by a single developer or a small team.

Short Lifespan and Development Cycle

Mobile applications have a short lifespan. Users have come to expect regular updates: from personal experience I found that more users will start to complain, regardless of the number of problems, if there has not been an update for over a month.

Given this short lifespan the development cycle has to be short which will likely result in more code smells. Maintenance and refactoring time will also be shorter. This makes it all the more important to have good refactoring tools.

A recent internet survey[3] found the average development time of a mobile application system to be 18 weeks with 10 weeks spent on the backend and only 8 weeks on the frontend (e.g. the Android application).

Regular Platform Updates

Table A.1 shows Android releases based on their API number. The API number of an Android release is increased by Google when new features are added that may require changes by the developer to support or backward compatibility by the system. When developing an Android application the developer specifies the target API for the application. This target API tells the Android system up to which API version the Android application is tested and can be expected to run correctly. If such an application is run on an Android device that has a version with a higher API number the system will enable backward compatibility behavior to ensure the application performs as expected.

An Android version with a new API is released every 92 days on average. Android developers are not required to keep their application targeting the latest release but are highly recommended to do so. As said above every major Android update has had UI changes that required application developers to update their application so it looked similar to the rest of the system UI. The Android SDK comes with a static analysis tool called Lint that gives a warning when the latest API is not targeted.

Limited Processing Power

Because processing power is limited on mobile devices power consuming actions like object creation should be kept to a minimum. This is especially true on the user interface thread which will freeze or delay the screen performance if methods that run on the UI thread take too long to finish.

One of the ways Android handles the limited processing power is recycling objects instead of creating new ones. The getView method of the Adapter class, which runs on the UI thread, has a recycled object as a parameter intended for reuse. The getView method of Adapters has the following signature:

```
View getView(int position, View convertView, ViewGroup parent)

    position: The position of the item ... whose view we want.
    convertView: The old view to reuse, if possible.
    parent: The parent that the view will be attached to.
```

For performance reasons an Adapter.getView method should use the recycled view when possible and have a correct implementation of the ViewHolder pattern.

Using the recycled views is done by checking whether the convertView parameter is null. If it is null then there are no recycled views available and a new view is created. If it is not null, the convertView is used as a new view and no object creation is required. When a view is no longer visible at the screen it is recycled. For example, when a list is shown on the screen only a limited number of items are visible at the same time. The user can scroll up or down to see new items while at the same time visible items are removed from the screen. The views that go out of the screen bounds will be recycled and given as a convertView to the next call to getView.

The getView also needs to implement the ViewHolder pattern which consists of using an instance of a class to hold references to the child views. Finding the view references consumes too much power to perform each time the list needs to be refreshed therefor the references are stored in a ViewHolder class that is saved in the parent view.

Both the usage of the recycled views and the ViewHolder pattern are implemented in the first part of the getView method. The end result of the first part is that the viewHolder instance is set. The second part consists of setting all the attributes of the view and it's child views, e.g. a label, an image, a color, etc.. An example of how this implementation looks in source code is shown below in Figure 2.3.

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

  ContactsViewHolder viewHolder;

  if (convertView == null) {
    convertView = layoutInflater.inflate(R.layout.contact_row,
        parent, false);

    viewHolder = new ContactsViewHolder();
    viewHolder.txName = (TextView) convertView.findViewById(R.id.
        tvName);
    viewHolder.txEmail = (TextView) convertView.findViewById(R.id
        .tvEmail);
    viewHolder.txPhone = (TextView) convertView.findViewById(R.id
        .tvPhoneNumber);

    convertView.setTag(viewHolder);
  } else {
    viewHolder = (ContactsViewHolder) convertView.getTag();
  }

  Contact contact = getItem(position);
  viewHolder.txName.setText(contact.getName());
  viewHolder.txEmail.setText(contact.getEmail());
  viewHolder.txPhone.setText(contact.getPhoneNumber());

  return convertView;
}

class ContactsViewHolder {
  TextView txName;
  TextView txEmail;
  TextView txPhone;
}
```

Figure 2.3: Example of a getView method following the Android guidelines.

This method does two things: the red code creates the UI while the blue code sets the values of the UI components.

Smaller Project Size

In a test set of of 20 open source Android applications in [15] it was found that on average the applications had 5,600 lines of code. In a cast study on source clones[9] two Java applications, ArgoUML and DNSJava, were analyzed. ArgoUML consists of 446 to 1538 classes with 45,000 to 200,000 NLOC and DNSJava consists of 55 to 179 classes with 5,000 to 25,000 NLOC.

External Libraries

Mobile applications rely to a large extent on external libraries: in a current research[15] the number of external calls was $^2/_3$ of all method calls. This observation is further described in Section 3.3.2.

Less Inheritance

In Mobile applications there is less inheritance [15] which points to the possibility of duplicate code and increased maintenance time. This observation is further described in Section 3.3.2.

High Interactive Applications

Mobile applications have more interaction caused by the presence of a touch screen.

2.3 Code Smells

2.3.1 Introduction

Code Smells describe patterns associated with bad design and bad programming practices. They point to areas in an application that could benefit from refactoring. Refactoring will improve the design of the application, make the application code easier to understand, help with maintainability and result in faster development [11]. Given the short development cycle refactoring is more useful for mobile developers.

In this section a definition of both the term Code Smell and a number of Code Smells is given. This section will also include assumptions on Code Smells that are more likely to occur in Android.

Mantyla [13] states the large number of code smells makes them difficult to understand. The code smells are divided into five groups:

Bloaters : Code that has grown so large it cannot be effectively handled (long methods, large classes).

Object-orientation abusers : Cases that do not fully exploit the power of object-oriented design (switch statements, temporary field).

Change preventers : Smells that hinder change or further development of the software.

Dispensables : Unnecessary code (lazy classes, duplicate code).

Couplers : Coupling-related smells (feature envy).

2.3.2 Code Smells Descriptions

This section gives a description of the code smells used in this thesis. Most of them come from Fowler [11] except the last one, Dead Code, which comes from Mantyla [13].

Large Class

The Large Class code smell is a class that has too many responsibilities, it is part of the Bloaters group. The solution according to Fowler is the Extract Class or Extract SubClass refactoring.

Long Method

The Long Method code smell is also part of the Bloaters group because the method has grown too large and is difficult to understand. According to Fowler most of the Long Method code smells can be resolved by applying the Extract Method refactoring.

Long Parameter List

This code smell is part of the Bloaters group since the parameter list has grown too large and is difficult to understand. Fowler gives three refactoring solutions: replace the parameter with a method call (Replace Parameter with Method), get the parameter data from an object (Preserve Whole Object) and move the parameter data into a new object and make the new object a parameter (Introduce Parameter Object).

Feature Envy

Feature Envy occurs when a method is more interested in data from another class then in data from its own class. Fowler suggests the obvious refactoring solution which is Move Method.

Switch Statements

The switch statement code smell is close to duplicate code. Often the same switch statement is found at multiple places. Fowler suggests using polymorphism by moving the different switch cases into subclasses. If there are only a few cases that affect a single method then polymorphism is overkill and the Replace Parameter with Explicit Methods refactoring can be used.

Duplicate Code

Duplicate code is according to Fowler *"number one in the stink parade"*. Luckily the solution is obvious: perform the Extract Method refactoring and replace all the duplicate code blocks with a method invocation to the new method.

Dead Code

The Dead Code code smell is not described in Fowler. It is found in [13] where it is classified as part of the Dispensables group. It is described as code that has been used in the past but currently is not used anymore. It hinders code comprehension and makes the structure less obvious.

2.3.3 Code Smells in Android

Activity classes

Activity classes in Android are responsible for a lot of different tasks what might result in a God Class smell. Looking at the MVC pattern an Activity is both the View and the Controller by design.

In the lifecycle methods of an Activity a lot of separate tasks have to be performed. The onCreate method to build the user interface, to get the references to every UI component used in the class, to set the listeners for the different events that may occur during the lifetime of the Activity, etc... The combination of these different tasks makes the lifecycle methods susceptible to the Long Method code smell.

Listener Methods

Android has many listener methods for events: the View class alone has already 10 listener methods. These methods have only one parameter which is the listener for an event, usually as an anonymous inner class. An example:

```
Button button1 = (Button) findViewById(R.id.button1);
button1.setOnClickListener(new View.OnClickListener()
{
  public void onClick(View v)
  {
    // do something when the button is clicked
  }
});
```

These listener method calls increase the number of lines in the onCreate method for the UI. Also the use of anonymous inner classes increase the likelihood of a Long Method code smell if there are multiple event listeners that need to be added or if there are a large number of UI components that need event listeners.

MenuItem clicks

In an Activity the method onOptionsItemSelected(MenuItem) is called when any of the menuitems is clicked. Other ways of getting notified when a menuitem is clicked is a listener on the MenuItem or specifying an Activity method to call in the XML.

This method is highly susceptible to the Switch Statements code smell. For example, when the menu is used to go to different screens then each Activity will have a switch statement to see which menuitem was clicked. Each of the activities will have almost the exact same code to handle the menuitem clicks.

This can be seen as a common code smell which Android developers will be very susceptible to because the Android API suggests that the onOptionsItem-Selected method is more efficient compared to the other options and easier in most situations.

Chapter 3

Related Work

3.1 Introduction

As explained at the beginning of this thesis there is not much research to be found on the subject of code smells in the mobile application domain.

3.2 Energy Code Smells

As discussed in Section 2.2.3 mobile applications need to be energy efficient. In [12] the authors explain how they try to improve the energy usage of mobile applications by searching for energy wasting patterns which they called energy code smells. Their focus is on applying reengineering techniques for removing energy code smells.

3.2.1 Example

In the paper they give examples of energy code smells, how they can be detected and be restructured. This example shows code of an open source application called GPS Print. Below is an altered and simplified version of the example given in the paper with in red and blue the code that is moved. Figure 3.1 shows the original code and Figure 3.2 shows the refactored code.

As can be seen from the code the refactoring is simple, the code contains two energy code smells: binding resources too early and releasing resources too late.

Binding resources too early

The first code smell is the binding resources too early code smell. It occurs when a resource, like the GPS receiver, is used before it is required for the application to function properly. In the refactoring example the requestLocationUpdates method invocation, which turns on the GPS receiver, is moved from the onCreate method to the onResume method which does not alter the behavior of the application but does turn on the GPS receiver at a later time.

```
public class GpsPrint extends Activity
    implements LocationListener {

  LocationManager lm;

  public void onCreate(Bundle savedInstanceState) {
    lm = (LocationManager) getSystemService(Context.
        LOCATION_SERVICE);
    lm.addGpsStatusListener(this);
    lm.requestLocationUpdates(LocationManager.GPS_PROVIDER, 1000,
         0, this);
  }

  public void onDestroy() {
    lm.removeUpdates(this);
  }
}
```

Figure 3.1: Original code of energy smell example.

```
public class GpsPrint extends Activity
    implements LocationListener {

  LocationManager lm;

  public void onCreate(Bundle savedInstanceState) {
    lm = (LocationManager) getSystemService(Context.
        LOCATION_SERVICE);
    lm.addGpsStatusListener(this);
  }

  public void onResume() {
    lm.requestLocationUpdates(LocationManager.GPS_PROVIDER, 1000,
         0, this);
  }

  public void onPause() {
    lm.removeUpdates(this);
  }
}
```

Figure 3.2: Refactored code of energy smell example.

Releasing resources too late

The second code smell is called the releasing resources too late code smell which is when a resource, like the GPS receiver, which consumes energy is used longer then required for the application to function properly. In the refactoring example the removeUpdates method invocation, which turns off the GPS receiver, is moved from the onDestroy to the onPause which does not alter the behavior of the application but does turn off the GPS receiver sooner.

Refactor Result

Moving the method that turns off the GPS receiver is the largest difference in energy consumption because the time between the onPause and onDestroy invocations can be very large while the time between the onCreate and the onResume is usually small. As can be seen from the lifecycle diagram in Chapter 2.2.1 the code before refactoring is not incorrect but it does consume more energy when it does not need to.

3.2.2 Other Energy Code Smells

[12] describes a number of other energy code smells some of which are not related to mobile applications alone. Below is a summary of the energy code smells described that are the most related to mobile applications.

Loop Bug

Loop bugs represent behavior where the application is repeating the same action over and over again without achieving the intended results. The example given in the paper is a server being down and the application that keeps trying to connect thereby using energy for the data connection. Another example is that the developer made a mistake causing an infinite loop, which if caused in a background thread keeps consuming CPU and using energy. The solution to this problem is to introduce a maximum number of iterations to perform the same action before stopping and reporting the problem to the user.

Using Expensive Resources

Using expensive resources is a code smell that occurs when there is a possibility to replace energy expensive resources with less energy consuming options, or if the expensive resource is used when it does not actually function. The example given for the first case is when only an approximate location is required and the expensive GPS location service is used. An example of the second case is having GPS active indoors or underground or having WiFi active without being in range of a WiFi station that can be connected to.

3.3 Minelli

[15] focusses at the first research question given in this paper: *'We want to study the source code of apps ... to understand if and how they differ from traditional software systems, and which are the possible implications for the maintenance of apps.'* Because these similarity this paper was a good starting point for the research in this thesis.

His analysis was performed with a selfmade tool called Samoa. Samoa is described as a web-based software analytics platform to analyze mobile applications from a structural and historical perspective. It shows a number of software metrics for the applications it analyzes and presents the metrics using different types of graphs. The metrics used include the Number of Packages, the Number of Classes, the Number of Methods, the Number of Internal Calls, the Number of External Calls, the Number of Core Elements, etc...

Samoa provides snapshot views (see Figure 3.3) which shows the structural properties of a mobile application. A circular view shows the application core and the external API method invocations.

The core elements are *"the entities specific to the development of apps (i.e., inheriting from the mobile platform SDK's base classes). In Android apps, they are specified in the manifest"*, where base classes are defined as Activities and Services. This narrow definition of base classes excludes a large number of entities that are also specific to the development of mobile applications:

- Custom user interface components, extending android.view.View
- Adapter implementations, extending android.widget.Adapter
- The other two application components:
  - content providers extending android.content.ContentProvider
  - broadcast receivers extending android.content.BroadcastReceiver
- The application singleton, extending android.app.Application
- Fragments, extending android.app.Fragment

Figure 3.3 shows a snapshot view of Alogcat (one of the applications analyzed, see Section 3.3.1).

The central part of the snapshot, labeled **a**, gives a visual representation about the size of Alogcat with regards to the lines of code. The core of the application consists of four elements: the yellow circle is the main activity called the LogActivity, the orange circle is another Activity and the two purple circles are services. As can be seen from the size the core accounts for almost half the size of the application. The none-core part of the application consists of 17 classes that are not shown; the none-core LOC is represented as the light blue circle while the light red circle is the core LOC.

The outer ring, labeled **b**, represents the number of external calls, the distance between the central part and this outer ring shows the number of internal calls. The outer ring is split into different pieces, the color of these pieces represents a different third party library. As can be seen from the snapshot most of the method invocations to third party libraries are to Android, in green, and Java, in orange. The light green outer circle represents the largest size of the application for all the revisions available. As can be seen there is a gap between the outer ring and the call ring which means that the latest revision of the application is not the largest.

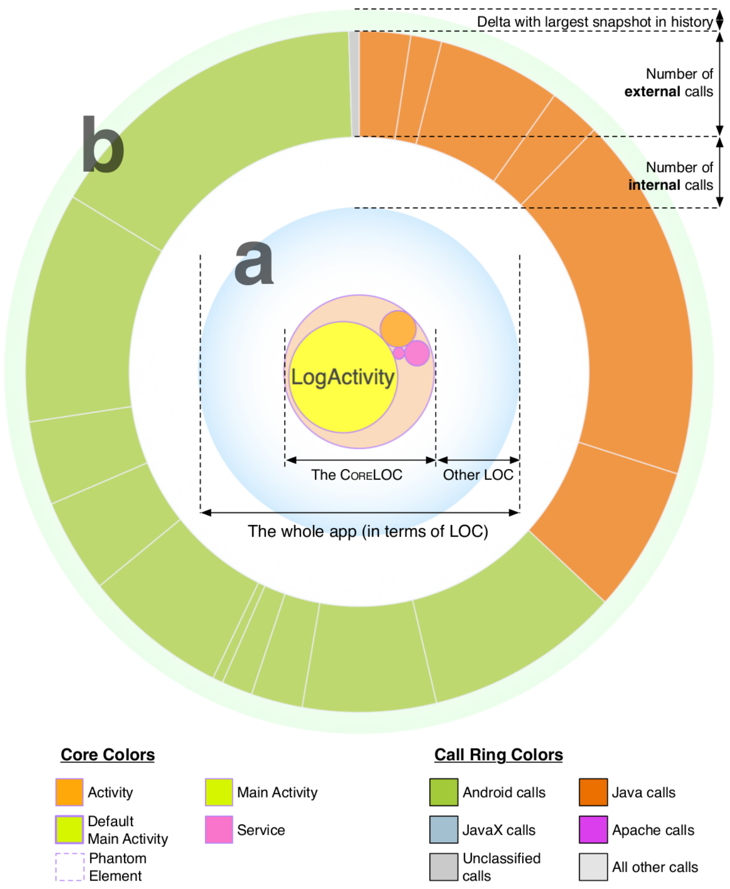


Figure 3.3: Example of a snapshot view made by Samoa with legenda.

3.3.1 Applications

Table 3.1 shows the applications that were analyzed with Samoa by Minelli. The table shows the application name, the LOC count, the Google Play Installs and the Google Play Rating (on a scale from 1 to 5).

3.3.2 Observations

The information that was gathered is used to give some insight into the structure of mobile applications. Minelli [15] presents a number of observations gathered during their analysis. The rest of this section is a summary of the observations.

Apps are smaller than traditional software systems

The first observation is that mobile applications have less lines of code compared to traditional applications. The average size of the applications analyzed is 5.6 kLOC with the smallest app, Searchlight, having less than 300 LOC. Figure 3.4 shows the snapshot of Searchlight. The app consists of two classes, the

Application Name	Lines Of Code Count	Google Play Installs	Google Play Rating
aLogcat	876	100.000+	4.6
Andless	2.372	100.000+	4.2
Android VNC	4.949	1.000.000+	4.3
Anstop	1.142	N/A	N/A
AppSoundmanager	1.605	50.000+	4.5
AppsOrganizer	8.321	1.000.000+	4.6
Csipsimple	20.777	100.000+	4.4
DiskUsage	4.749	50.000+	4.7
MythDroid	6.114	N/A	N/A
MythMote	1.593	10.000+	4.6
Open GPSTracker	9.754	100.000+	4.2
OpenSudoku	3.813	1.000.000+	4.6
ReplicaIsland	14.192	1.000.000+	4.2
RingDroid	3.516	10.000.000+	4.6
Search Light	272	100.000+	4.7
Share My Position	468	10.000+	4.6
SipDroid	14.019	500.000+	4.0
Solitaire	3.343	10.000.000+	4.3
Zirco Browser	6.429	10.000+	3.8
Zxing	3.407	50.000.000+	4.3

Table 3.1: List of applications analyzed by Minelli.

Main Activity (in yellow, the core) and a none-core class (the light blue area). Together the two classes contain 272 LOC.

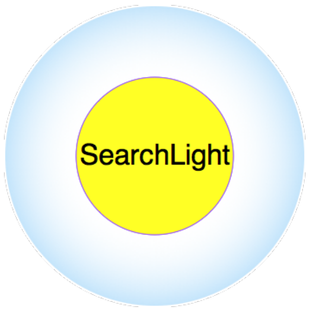


Figure 3.4: Snapshot of Searchlight, smallest app analyzed.

The main reason for the small size of source code is that many applications have a small number of functions that needs only a few classes to implement them.

Apps are inherently complex, mostly because they rely on third-party libraries

Minelli [15] found that 2/3 of method invocations were to third-party libraries. This observation was surprising to me which is why I chose it as an initial analysis goal to verify this claim. This analysis is given in Subsection 3.3.3

In many of the apps that were analyzed over 75% of all method calls were external calls which means the number of internal calls were small. An example of this is shown in Figure 3.5 which represents the call ring of Share My Position.

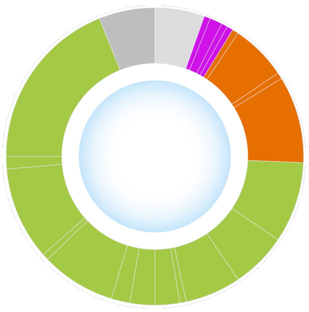


Figure 3.5: Call ring of Share My Position.

As said before the thickness of the outer ring represents the number of third-party calls and the thickness of the white ring between the outer ring and the blue ring represents the number of internal calls. Because apps rely strongly on third-party libraries both the source code of the application needs to be looked at and the behavior of the third-party library needs to be understood to comprehend how an application works. This complicates both program comprehension and maintenance.

The size and complexity of apps grow in correlation with the addition of third-party method invocations

They found high values of correlation between the number of third-party method invocations and two other metrics, McCabe's cyclomatic complexity number with an average correlation value of 0.82 and number of LOC with an average correlation value of 0.84.

The use of inheritance is essentially absent in apps

By looking at the average values of two software metrics related to inheritance they observed that there was very little inheritance to be found for the applications analyzed. The average value for Average Hierarchy Height was only 0.09 and the average value for Average Number of Derived Classes was 0.19 which compared to traditional Java applications is very low. According to Minelli a possible reason could be that the applications analyzed are much smaller than traditional Java applications thereby reducing the need for inheritance. Another reason could be that many of the mobile applications are not developed in a systematic way which can result in badly structured code and duplicate code smells.

According to [10] maintaining a flat system requires 20% more effort than an analogous system using inheritance. This means maintaining mobile applications is made more difficult because of the lack of inheritance.

Development guidelines are often ignored

As described above in Section 2.2.1 an Android application normally has only one main activity. This is actually stated in the Android guidelines: *"an app consists of multiple activities loosely bound to each other. Typically, one activity is specified as the main activity, which is presented to the user when launching the application for the first time"*. It was found that multiple applications defined more than one main activity. These main activities represent multiple entry points into the application which increases the complexity and complicated their comprehension.

Some apps are only composed of the core

They found that on average half of the LOC of an application represents core classes. Nearly 25% of the applications they analyzed consisted for 70% of core classes. As an example they give the Andless application which has about 2300 LOC and only 60 of them are not in core classes.

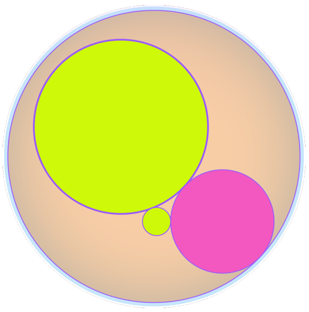


Figure 3.6: Snapshot of Andless

The main activity of Andless has 1700 LOC and is responsible for a lot of different tasks:

- Drawing the UI
- Starting & stopping the music
- Searching music on the file system
- Parsing playlists
- Handling cue files

This makes the main activity of Andless a God Clas which represents all kind of maintenance problems.

3.3.3 Initial Analysis

One of the observations was that roughly 2/3 of all the method invocations in an Android application is a method invocation to a third party library. I have chosen half of the applications in the Minelli paper to analyze their method invocations and see if the claim is correct.

Process

Figuring out whether the methods invoked are from third party libraries or not was done by building a small Eclipse plugin. The plugin creates an AST of the project classes using the AST parser of the Eclipse JDT library. The AST consists of nodes that refer to a construct in the source code. One type of node is referring to method invocations. The AST parser resolves the references of the method invocation include the package and class name of the method that is invoked. The package determines whether the method invocation is to android (the package starts with android), java (the package starts with java or javax), the project itself (if the package is in the project package list) or other (all remaining packages, e.g. third party libraries).

Results

The results of the method invocation analysis is given below in Table 3.2 and in Figure 3.7. The last column of the table, External, is the sum of the Android, Java and Library calls.

	Android	Java	Library	Project	External
SipDroid	1743	2073	0	4462	3816 / 8278 (46%)
aLogcat	172	97	0	121	269 / 390 (68%)
AppsOrganizer	975	804	0	1285	1779 / 3064 (58%)
DiskUsage	580	509	0	708	1089 / 1797 (60%)
MythDroid	1668	951	148	1382	2767 / 4149 (66%)
MythMote	623	126	0	228	749 / 977 (76%)
OpenSudoku	1072	369	35	643	1476 / 2119 (69%)
ReplicaIsland	672	753	32	6057	1457 / 7514 (19%)
RingDroid	636	298	4	283	938 / 1221 (76%)
Solitaire	368	76	0	810	444 / 1254 (35%)
Total	8509	6056	219	15979	14784 / 30763 (48%)

Table 3.2: The number of method invocations by application split up for packages.

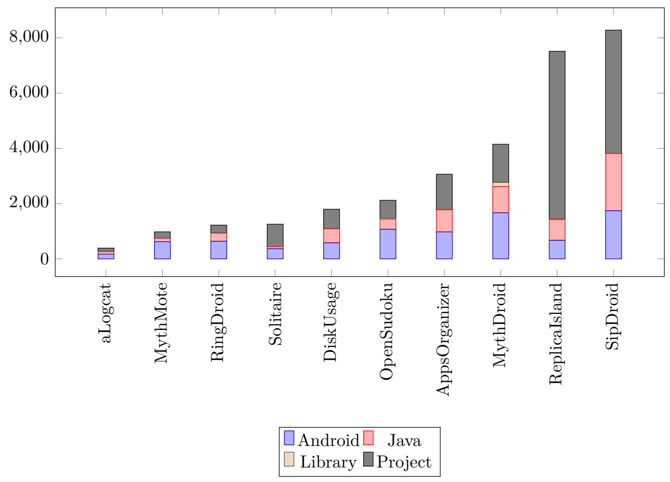


Figure 3.7: The number of method invocations by application split up for packages.

The table also shows the total number of calls and the percentage of external calls. The average percentage of external calls for the 10 applications analyzed is 48% which is close to the 67% found in the claim. The difference might result from the fact that only half of the applications from the original study were analyzed or perhaps different revisions were analyzed.

Third-Party Libraries

The minor difference in the results is less interesting compared to the fact that they claim 67% of the calls is to third party libraries whereas Table 3.2 shows external calls including Android and Java calls. So in their paper Java API libraries and the Android API libraries are considered third party libraries. Given this definition a Java or Android application can not be made without third party libraries.

My definition of third party libraries is more like one found online: *'... a third-party software component is a reusable software component developed to be either freely distributed or sold by an entity **other than the original vendor of the development platform**.'* Both the Android API libraries and the Java API libraries that come with Android are distributed by the original vendor of the development platform. I believe the Android API libraries and the Java API libraries should be described as system libraries because they are part of the Android Operating System.

3.4 Fowler

[11] contains an introduction into refactoring, it describes 22 code smells and consists for a large part of a catalog of refactorings. Many of the refactorings given by Fowler are methods on how to get from where the code is with the code smell to a given code pattern without the code smell. One of the reasons to perform refactoring is that it can make object-oriented code easier to understand. Code that is easier to understand results in shorter development time because complex software code hinders software development.

Section 2.3.2 describes the 7 code smells used in this paper and the Fowler refactorings on how to solve them. The rest of this section will be about those refactorings. Some of these refactorings consists of performing other refactorings, these can be found in [11].

3.4.1 Extract Class

The Extract Class refactoring is used to solve the Large Class code smell. The change consist of splitting one (large) class into two smaller ones. This refactoring consist of choosing which responsibilities of the large class are moved to a new class and using the Move Field and the Move Method refactorings to move the fields and methods related to the responsibilities and testing the result.

3.4.2 Extract SubClass

The Extract SubClass refactoring can also be used to solve the Large Class code smell. The choice between Extract Class and Extract SubClass is based on whether the class includes functionality only used in some instances and not in others. The refactoring consists of creating a new subclass and using the Push Down Method and Push Down Field refactorings to move features into the subclass and test the result.

3.4.3 Extract Method

The Extract Method refactoring is used to solve the Long Method code smell and the Duplicate Code code smell. The refactoring consists of creating a new method, moving part of the code into that new method, adding all variables referenced in the code as parameters to the new method, adding a call to the new method where the code used to be and test the result.

3.4.4 Replace Parameter with Method

The Replace Parameter with Method refactoring can be used to solve the Long Parameter List code smell. This refactoring can be performed when the result of a method invocation is passed to a method call while the method invocation can also be done inside the method that is called. The refactoring consist of replacing references to the parameter inside the method to references to the method call, perform the Remove Parameter refactoring and test the result.

3.4.5 Preserve Whole Object

The Preserve Whole Object refactoring can be used to solve the Long Parameter List code smell. This refactoring can be performed when values from an object are passed as parameters in a method call. The refactoring consists of creating a new parameter for the object where the data comes from, determine which parameters should be obtained from the whole object and replace the references to those parameters with calls to methods of the object, delete the obsolete parameters in the method and method calls and test the result.

3.4.6 Introduce Parameter Object

The Introduce Parameter Object refactoring can be used to solve the Long Parameter List code smell. This refactoring can be done when there is a group of parameters that belong to each other. The refactoring consist of creating a new class that represents the group of parameters, adding the new class to the parameter list, moving the parameters from the parameter list into the new class and replacing all references to the removed parameters with references to the new class parameter.

3.4.7 Move Method

The Move Method refactoring can be used to solve the Feature Envy code smell. This refactoring can be done when a method is using or used by more features of another class than the class on which it is defined.

3.4.8 Replace Parameter with Explicit Methods

The Introduce Parameter Object refactoring can be used to solve the Switch Statements code smell. The refactoring can be done when a method runs difference code depending on the value of an enumerated parameter. The refactoring consists of moving the code in the different cases into new methods and directly calling these methods.

3.5 Eclipse Parsing Tools

3.5.1 Introduction

Part of this thesis consists of creating an Eclipse plugin that looks for patterns in a Java project. These patterns are described in Chapter 6. This section describes the creation of an Eclipse plugin that will use the Eclipse JDT Core to both analyze and refactor Java code. A great resource for this is the PhD. dissertation [18] of the author of JDeodorant which is one of the tools used in the analysis described in Chapter 4. Chapter 6 of his dissertation describes the JDT Core that JDeodorant uses to first search for code smells and later refactor the code block to remove the code smell.

3.5.2 Java Elements

Eclipse JDT has two different representations for Java code. The first one is the Java Model which is a high level representation. The nodes of the Java Model tree are shown in Figure 3.8.

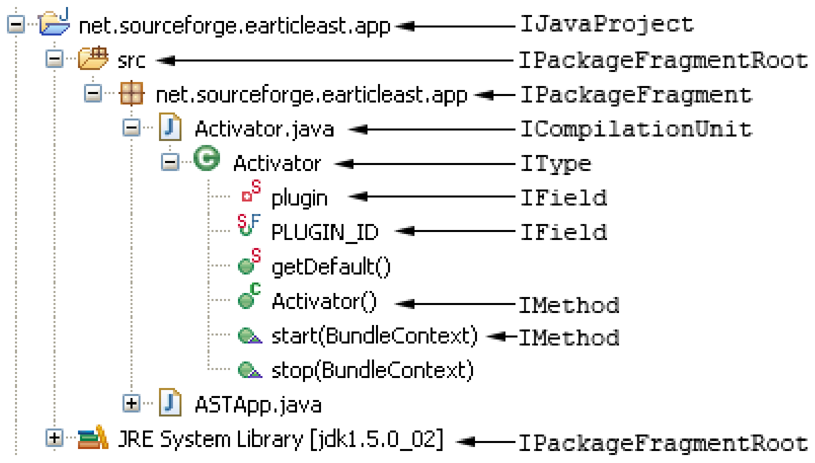


Figure 3.8: Eclipse Java Package Explorer

The lowest node types represent the methods and fields. This means the Java Model tree does not give access to the statements. This is what the low level representation does.

Full access of the Java source code is provided by the Abstract Syntax Tree (AST). The AST is build from an ICompilationUnit which is the Java Model representation of a Java source file. The AST has a CompilationUnit as the root node. All nodes are subclasses of the ASTNode and are grouped in four abstract superclasses:

- BodyDeclaration: all body declarations (classes, methods, types, enums, ...)
- Type: all types (primitives, arrays, including generics, ...)
- Statement: all statements within method bodies (for, if, switch, ...)
- Expression: all expressions within statements (method invocations, field access, assignments, ...)

The AST can also be used to refactor the Java source code by adding, changing and removing nodes in the AST.

Chapter 4

Research Framework

4.1 Introduction

This chapter describes which applications were analyzed and what tools were used to analyze them. It also explains which code smells were looked for and how the tools found them.

4.2 Applications

4.2.1 Commercial Applications

Part of my thesis study consisted of analyzing a small collection of commercial applications. I was given the opportunity to do so at a mobile application development company in Amsterdam. Based on the project size and development methods I chose 4 out of their 12 Android applications as part of my research subject. The names of the applications are anonymized because these applications were developed for clients and are still in use. The applications are OO, TC, WW and LS. OO was developed using the Extreme programming development method. TC was developed by an intern co-worker; the development method is unknown. WW and LS were both developed using the waterfall development method.

4.2.2 Open Source Applications

As previously explained in Section 3.3.1 Minelli used the open source application store FDroid as a source for his research subject. The main analysis used the same ten FDroid applications that were used to verify the third party library claim.

4.2.3 Own Test Application

To test the tools I used for my research framework I have made a simple app with deliberate code smells.

4.3 Code Smell Tools

4.3.1 Introduction

The main part of this thesis consisted of finding code smells in Android projects to determine whether code smells occur more often in Android related code. This section consists of the selection process of the tools to find code smells and a description of the selected tools.

Searching for tools for code smells in Android applications it became clear that there were not many tools available yet. Since most of the source code of an Android application consists of Java code the search was extended to include tools that find code smells in Java applications. All of the tools described in this chapter with the exception of Lint are not developed for Android application analysis.

Lint is the only analysis tool found, that was developed specifically for analysis of Android applications. It is a tool that is part of the Android SDK; it performs most of the functions an Android developer may require of an Android source code analyzing tool.

The following tools were available:

- JDeodorant
- Lint
- Checkstyle
- PMD
- UCDetector

A combination of these tools are recommended by Android developers so this will be a good starting point.

In [8] some of the tools above are mentioned and three others:

- iPlasma
- Decor
- Stench Blossom

Basic properties of these tools are given in Table 4.1.

4.3.2 JDeodorant

JDeodorant [18] is an Eclipse plugin first released in late 2007. It is made by Nikolaos Tsantalis and identifies bad smells; it is also able to resolve smells by applying appropriate refactorings. It can handle the God Class, Long Method, Type Checking and Feature Envy code smells. Support for the code smell Duplicate Code is under development.

The output is saved to a text file after pressing a button and entering a filename. The output contains the name of code smell found with the class

Tool Name	Eclipse Plugin / Standalone	Supported Languages	Supports Refactoring	Code Available
JDeodorant [18]	Eclipse Plugin	Java	Yes	Yes
Lint [4]	Eclipse Plugin, Standalone	Java, XML	Partially	Yes
Checkstyle [2]	Eclipse Plugin, Standalone	Java	No	Yes
PMD [5]	Eclipse Plugin, Standalone	Java	No	Yes
Decor [16]	Plugin of Ptidej (part of tool suite)	Java	No	No
iPlasma [14]	Standalone	Java, C++	No	No
Stench Blossom [17]	Eclipse Plugin	Java	No	Yes
UCDetector [7]	Eclipse Plugin	Java	No	Yes

Table 4.1: Basic properties of the code smell tools

name and method name that contains the smell. Pressing a button for each project would take too long so I requested the source code from the author. After receiving the source code and making some changes the output could be written to a XML file.

God Class

God Class code smells are found by looking for Extract Class refactoring opportunities using the Eclipse framework.

Long Method

Long Method code smells are found by searching for Extract Method refactoring opportunities using slicing techniques.

Type Checking

JDeodorant distinguishes between two different Type Checking code smell cases. The first one is where the conditional code fragment is a switch statement and the second one is where it is an if / else if structure.

Feature Envy

Feature Envy code smells are found by looking for Move Method refactoring opportunities: it searches for methods that will use less foreign resources when moved to another class.

4.3.3 Lint

Lint is a static code analysis tool released as part of the Android Developer Tools (ADT), the Android Eclipse plugin. It is developed by Google and released in late 2011 along ADT 16; the version used in this thesis is ADT 21. It is both a standalone application and an Eclipse plugin that analyzes Android project source files. So far this is the only static analysis tool specificly targeting Android. The description of the tool does not mention checking for code smells but since this is the only tool specificly aimed for Android it is included in the analysis to see if it really does not include any functions for code smell checking. The output is written to disk into a XML or text file, the contents can be seen in C.1.

4.3.4 PMD

PMD is a tool that analyzes Java source code. It looks for potential problems or bugs like dead code, empty try/catch/finally/switch statements, unused local variables or parameters, and duplicated code. PMD is able to detect Large Class, Long Method, Long Parameter List and Duplicated Code smells by looking at metrics for the first three and comparing lines of code for the last one. In a settings file the minimum thresholds values for the metrics used by PMD can be set. The output of PMD is an XML file that contains the location (source file name with line number) and the type of the code smell that was found.

There is no official definition of what PMD stands for but on the PMD wikipedia page it says the following:

> *it has several unofficial names, the most appropriate probably being Programming Mistake Detector* [6]

Large Class

The best option PMD has for searching Large Class code smells is the LOC of the class. This is not how Fowler describes this code smell (number of variables) but it is the option that best fits the actual definition. PMD looks for the Large Class code smell using the ExcessiveClassLength rule. The minimum LOC of the class which is considered a Large Class code smell is 1000 by default.

Long Method

The Long Method code smell is found using the ExcessiveMethodLength rule. It looks for the LOC of the method. The minimum LOC of the method which is considered a Long Method code smell is 100 by default.

Long Parameter List

The Long Parameter List code smell is found using the ExcessiveParameterList rule. The minimum number of parameters it will report is 10 by default.

Duplicate Code

The Duplicate Code code smell is found by looking for a sequence of lines that appears at multiple places.

4.3.5 Checkstyle

Checkstyle is a static analysis tool first released in 2001. In this thesis version 5.6 is used. Its main function is to help developers write Java code that adheres to a coding standard. A number of code smells are related to the properties that the Checkstyle checks look for: the Long Method code smell can be found using the MethodLength module of Checkstyle which looks for long methods and constructors. Other code smells that Checkstyle can find are Large Class, Long Parameter List and Duplicated Code. Metrics like how many lines of code are required to be considered a Long Method code smell are configurable by settings a maximum allowed threshold value.

Checkstyle uses a parser generator to construct an abstract syntax tree (AST). The resulting AST represents blocks of code consisting of classes, methods, and other control structures. Checkstyle provides full access to this AST which allows others to extend Checkstyle by writing checks themselves.

The output of Checkstyle is a XML file containing the location (source file name with line number) and the type of the code smells that were found.

Long Method

The Long Method code smell is found using the MethodLength check, it looks for the LOC of a method. The maximum allowed value is 150 by default which means a method having 151 or more LOC is considered a code smell.

Long Parameter List

The Long Parameter List code smell can be found using the ParameterNumber check. The maximum number of parameters allowed is 7 which means 8 or more parameters are considered a code smell.

Duplicate Code

The Duplicate Code code smell is found by using the StrictDuplicateCode check. It searches for a sequence of lines that appears at multiple places. The maximum length of a sequence is 12 by default.

4.3.6 Decor

Decor is a platform for software analysis. One of the functions is finding anti patterns (or code smells). The tool can find six code smells. Of all the tools listed Decor is the only one that is not freely available. It is part of a tool suite which is available upon request.

4.3.7 iPlasma

iPlasma is an integrated environment for quality analysis of object-oriented software systems. It includes support for all the necessary phases of analysis: from model extraction to high-level metrics-based analysis. It is able to detect code duplication. It can be used to analyze large open-source systems like Eclipse (which at the time had 1.4 million lines of code).

It does not appear to be possible to export any specific code smell related data. An image of design flaws can be exported but no information is given about the image. An 'Interpretation of the Overview Pyramid' can also be exported from this tool. It gives metrics such as Cyclomatic Complexity, Lines Of Code, Number Of Methods, Number Of Classes, Number Of Packages, Number of distinct method-calls.

It also gives a summary which contains an indication of the Large Class and Long Method code smells. Below is an example of this summary for the OO application:

- Classes tend to:
  - be rather *large* (i.e. they define many methods);
  - be organized in rather *fine-grained packages* (i.e. few classes per package);
- Methods tend to:
  - be rather *long* yet having a rather *simple logic* (i.e. few conditional branches);
  - call *few methods* (low coupling intensity) from *few other classes* (low coupling dispersion);

4.3.8 Stench Blossom

Stench Blossom is an Eclipse plugin that provides an interactive visualization environment to give programmers a quick and high level overview of the code smells in their code and their origin. The plugin works inside the Eclipse code editor and only shows code smells for the source code that is currently open.

The plugin shows the code smell results visually with the help of as a set of petals on the right side of the source code. The size of a petal is proportional to the strength of the code smell it refers to. The tool can detect eight code smells: Data Clumps, Feature Envy, Instanceof, Large Class, Long Method, Message Chains, Switch Statements and Typecast.

It appears to only function on old Eclipse releases. It works on Eclipse 3.3 while the Android Development Tools is bundled with Eclipse 4.2. Trying to get ADT to function in Eclipse 3.3 failed and getting Stench Blossom working on Eclipse 4.2 also failed.

4.3.9 UCDetector

UCDetector can detect three different problems with Java Code: unnecessary code, code where the visibility can be changed and methods or fields which can be final. The code had to be changed to automate the process just like with JDeodorant.

The output of UCDetector is a XML file containing the package name, class name and method names of the problems it finds.

Dead Code

UCDetector finds Dead Code code smells that are called unnecessary code problems.

4.3.10 Samoa

Samoa itself was not included because the authors did not provide access to the source code and I was not allowed to share the source code of the commercial applications.

4.3.11 Supported Code Smells

Table 4.2 gives an overview of what code smells the tools support. Those marked with a X are described in the previous sections while the ones marked with a star indicate the tool did not work as expected or was not available.

	JDeodorant	**Lint**	**Checkstyle**	**PMD**	**Decor**	**iPlasma**	**Stench Blossom**	**UCDetector**	**Total**
God Class	X								1
Large Class				X	*	*			3
Long Method	X		X	X	*		*		5
Long Parameter List			X	X	*				3
Feature Envy	X					*	*		3
Switch Statements	X						*		2
Duplicate Code			X	X		*			2
Dead Code				*				X	2
Total	4	0	3	5	3	3	3	1	

Table 4.2: Code smell support

As can be seen most tools support the Long Method code smell, followed by Large Class, Long Parameter List and Feature Envy.

Lint does not support any of the code smells mentioned above. It does find problems in Android layout files, e.g. when a view can be removed without altering the result or when two views can be merged while the UI remains the same. These might be considered layout smells: the current structure is not wrong but changing it to what Lint suggests will improve the structure. The Lint results for OO and SipDroid are shown in Table C.1.

4.3.12 Selection

Tools

Not all of the tools that were selected based on their supported code smells were actually usable: Decor as said before is not freely available, iPlasma does not provide any useful information, Stench Blossom is not compatible with the newer versions of Eclipse that Android requires and Lint does not find any code smells. Only JDeodorant, Checkstyle, PMD and UCDetector were used as analysis tools. All 4 tools have enough information in their output to find the package name, class name and method name of the code smells found.

Code Smells

Out of the remaining tools at least two support the following code smells: Long Method, Large Class (assuming God Class and Large Class are similar) and Long Parameter List. Feature Envy is supported by only one of the remaining tools but by three of the initial tools so it was included. And UCDetector is included to have at least one code smell not described by Fowler, instead it comes from [13].

This means the analysis looked for the Long Method, Large Class, Long Parameter List, Feature Envy and Dead Code code smells.

4.3.13 Tool Validation

To validate the tools selected an analysis was done on a subset of the applications and these results were verified manually to validate the code smells found. The tool validation was performed a number of times to find the tool parameters (e.g. PMD and Checkstyle method and class length thresholds) that give the best results.

The validation of the tools and a discussion of the results can be found in Appendix B, a summary is given in Table 4.3. The table contains the number of code smells where I agree with the findings of the tool and the total number of code smells found by the tool.

As can be seen from the table I agreed with all code smells found except for one smell found in OO: a method of only 3 lines of code which JDeodorant claimed was a Long Method code smell.

	OO	**SipDroid**	**Custom App**
JDeodorant			
God Class	5 / 5	3 / 3	1 / 1
Long Method	9 / 10	9 / 9	3 / 3
Checkstyle			
Long Method	15 / 15	10 / 10	3 / 3
PMD			
Long Method	15 / 15	10 / 10	3 / 3
Large Class	2 / 2	3 / 3	1 / 1

Table 4.3: Summary of tool validation results with agreed and total code smell count

While performing the tool validation I found something interesting about JDeodorant. As described in Section 4.3.2 JDeodorant looks for God Class code smells by finding opportunities to perform the Extract Class refactoring of Eclipse. The result is that the code shown below is considered a God Class because the two fields and their corresponding methods can be separated.

```
public class GodClass {
  private String str1;
  private String str2;
  private void loadString1() {
    str1 = "abc";
  }
  private void loadString2() {
    str2 = "abc";
  }
}
```

This is a good example of the difference between a God Class of JDeodorant and a Large Class of PMD.

Chapter 5

Analysis

5.1 Introduction

In Chapter 4 the applications of this analysis and the tools to be used were described. This Chapter will describe the details and the results of the analysis.

5.1.1 Android Relevance

During the analysis methods or classes were classified as specific to mobile development or not. In Section 3.3 Minelli classifies core elements as *"classes that inherit from the mobile platform SDK's base classes"* with the base classes being defined as Activities and Services. In this thesis this narrow definition is extended to include all other Android components like Fragments and Adapters (see 2.2.1). In the extended definition core classes are described as *classes that inherit from the mobile platform SDK's classes* where the mobile platform SDK's classes are all classes in the Android API library.

5.1.2 What was Analyzed

In the analysis six code smells (Large Class, Long Method, Feature Envy, Type Checking, Long parameter List and Dead Code) were looked at using the four tools described in Chapter 4. The output included the location of the code smell and whether the code smell was found in a core class or not. We calculated the NLOC and which percentage of it is related to Android.

5.1.3 Expected Results

- Section 2.3.3 discussed why the Large Class, Long Method and Switch Statement code smells are more likely to occur in core classes.
- There are no indications that the Feature Envy and Dead Code code smells are more likely to occur in core classes.
- Long Parameter List smells are probably less likely to occur in core classes because of the many methods that need to be overridden (e.g. the lifecycle methods in Activities).

- Type Checking smells are probably more likely to occur in core classes because most application screens have a menu. In Android menu item clicks are usually handled by a method containing a switch statement to check which menuitem was clicked.

The tool validation phase gave some insights into which code smells can be expected. A large number of Large Class and Long Method code smells were found in a small piece of code. No Long Parameter List code smells were found.

5.2 Process

Below is a description given of the analysis process. It includes a description of how the tools work and how it is determined whether the code smell is related to mobile development or not.

JDeodorant and UCDetector required some modifications to simplify getting the results, as described in Section 4.3.2 and 4.3.9, respectively. After Eclipse was started from the command line, the plugins started their analysis and the output was written to disk. PMD and Checkstyle were also started from the command line and their output was also written to disk. During the measurements we also calculate the total NLOC and the total NLOC found in core classes.

The next step in the analysis was to figure out whether the code smells found were specific to mobile application development. For JDeodorant this was done by another modification of JDeodorant; an AST was created to build a list of classes considered to be core classes. The output of the tools both included the code smells and whether they were in a core class or not.

At the end of the process we know the following:

- The number of code smells in core classes: **CoreCS**
- The total NLOC in core classes: **CoreNLOC**
- The number of code smells in non-core classes: **NonCoreCS**
- The total NLOC in non-core classes: **NonCoreNLOC**

5.3 Results

5.3.1 JDeodorant

The JDeodorant results of the analysis can be found in Appendix C.2.

The results of the smallest commercial application, LS, show that out of the 17 code smells found in the application only 1 is not in a core class. This is to be expected of this small application given that it consists of a number of Activity classes, a few model classes and two util classes.

5.3.2 Checkstyle

Checkstyle results are in Appendix C.3.

5.3.3 PMD

PMD results are in Appendix C.4.

Most of the results of Checkstyle and PMD are the same which can be explained by having the same threshold values but there is a small variation in the Long Method results which might be caused by Checkstyle including the method signature in the method length whereas PMD only includes the method body.

5.3.4 UCDetector

UCDetector results are in Appendix C.5.

5.3.5 NLOC

As described above we also calculate the total number of NLOC, the number of NLOC in core classes and the percentage of CoreNLOC ($\frac{CoreNLOC}{CoreNLOC+NonCoreNLOC}$). These metrics are shown in Table 5.1.

Project	**CoreNLOC**	**NonCoreNLOC**	**CoreNLOC %**
OO	9727	8397	53 %
LS	1191	553	68 %
TM	3115	2690	53 %
WW	4499	915	83 %
SipDroid	5254	30929	14 %
aLogcat	719	659	52 %
AppsOrganizer	3013	16741	15 %
DiskUsage	1925	5652	25 %
MythDroid	9361	7135	56 %
MythMote	2348	1058	68 %
OpenSudoku	4216	3411	55 %
ReplicaIsland	3976	21345	15 %
RingDroid	3191	2636	54 %
Solitaire	1011	3371	23 %
Total	53546	105492	33 %

Table 5.1: CoreNLOC and NonCoreNLOC for all projects.

5.4 Code Smells in Core Classes

5.4.1 Introduction

From the analysis results we derived the following metrics: **CoreCS**, **CoreNLOC**, **NonCoreCS** and **NonCoreNLOC**.

Using these metrics we calculate the number of code smells found per 1,000 lines of code by calculating $\frac{CoreCS}{CoreNLOC/1000}$ and $\frac{NonCoreCS}{NonCoreNLOC/1000}$.

5.4.2 Results

Table 5.2 and Table 5.3 show the number of code smells per 1,000 lines of code for all tools and code smells. Table 5.4 shows the results grouped by code smell.

	Long Method						Large Class			
	JDeodorant		PMD		Checkstyle		JDeodorant		PMD	
	Core	Non-Core	Core	Non-Core	Core	Non-Core	Core	Non-Core	Core	Non-Core
OO	5.8	4.5	10.3	4.4	10.8	4.6	2.5	2.5	2.9	2.4
LS	9.2	1.8	10.1	3.6	11.8	3.6	2.5	0	4.2	3.6
TM	4.8	4.5	6.4	5.9	6.7	5.9	1.6	2.2	1.6	2.6
WW	9.1	1.1	10.2	1.1	10.4	1.1	1.8	4.4	3.1	2.2
SipDroid	8.2	6.8	7.2	2.7	7.2	2.7	1.5	1.5	2.1	1.8
aLogcat	7.0	4.6	5.6	4.6	7.0	4.6	4.2	4.6	1.4	1.5
AppsOrganizer	2.7	4.0	5.0	0.8	5.0	1.0	2.7	1.6	2.0	1.6
DiskUsage	5.7	7.1	5.7	5.7	5.7	5.8	2.1	3.0	2.6	1.6
MythDroid	5.1	4.2	7.9	5.3	8.1	5.5	2.2	2.1	2.7	2.7
MythMote	6.0	1.9	5.5	10.4	6.8	10.4	2.6	2.8	2.1	2.8
OpenSudoku	6.6	6.2	5.9	3.5	6.2	3.8	2.1	3.8	3.3	2.1
ReplicaIsland	3.8	9.1	3.8	6.5	4.0	6.5	1.5	2.2	2.0	1.7
RingDroid	6.9	6.1	6.0	5.7	6.0	5.7	1.6	1.5	1.3	2.3
Solitaire	11.9	6.8	11.9	3.9	11.9	4.2	2.0	2.4	2.0	2.1
All Apps	6.1	6.2	7.5	3.9	7.9	4.0	2.1	2.0	2.5	1.9

Table 5.2: Number of code smells found per 1,000 LOC for Long Method and Large Class.

	Long Parameter List				Feature Envy		Type Checking		Dead Code	
	PMD		Checkstyle		JDeodorant		JDeodorant		UCDetector	
	Core	Non-Core	Core	Non-Core	Core	Non-Core	Core	Non-Core	Core	Non-Core
OO	0.1	0.1	0.1	0.1	1.2	0.6	1.1	0	1.7	2.5
LS	0	0	0	0	1.7	0	0	0	0	10.8
TM	0	0	0	0	2.2	0.4	0.3	0.7	1.9	6.3
WW	0	0	0	0	4.2	0	0	0	1.6	8.7
SipDroid	0.2	0.4	0.2	0.4	0.4	1.2	1.9	0.2	2.5	1.6
aLogcat	0	0	0	0	8.3	3.0	0	0	1.4	3.0
AppsOrganizer	0	0	0	0	0.7	0.7	0.7	<0.1	2.3	6.2
DiskUsage	0	1.1	0	1.1	1.6	1.4	0	0.5	2.6	0.7
MythDroid	0	0.1	0	0.1	0.1	0.3	0.3	0	2.7	3.8
MythMote	0	0	0	0	1.3	3.8	0	0	3.4	0.9
OpenSudoku	0	0	0	0	1.7	2.6	0.9	1.5	0.9	2.1
ReplicaIsland	0	0.7	0	0.7	0.3	1.7	2.0	0.9	0.8	1.3
RingDroid	0	0	0	0	0	0	0.3	0.4	0.9	0.8
Solitaire	0	0	0	0	2.0	1.8	5.9	2.4	0	1.5
All Apps	<0.1	0.3	<0.1	0.3	1.3	1.2	0.9	0.4	1.8	2.7

Table 5.3: Number of code smells found per 1,000 LOC for Long Parameter List, Feature Envy, Type Checking and Dead Code.

5.4.3 Code Smells

Code Smell	Core	Non-Core
Long Method	7.2	4.7
Large Class	2.3	2.0
Long Parameter List	<0.1	0.3
Feature Envy	1.3	1.2
Type Checking	0.9	0.4
Dead Code	1.8	2.7

Table 5.4: Number of code smells found per 1,000 LOC grouped by Code Smell.

The results in Table 5.4 give an overview of the results grouped by code smell.

Long Method

It shows that the Long Method code smell is almost twice as likely to occur in core classes which was expected.

Large Class

It also shows that the Large Class code smell is almost as likely to occur in core classes as in non-core classes which was not expected. A possible reason for this could be that the number of Activity classes compared with the total number of classes is small.

Long Parameter List

The Long Parameter List code smell is almost non-existent in core classes with less than 1 Long Parameter List code smell per 10,000 LOC.

Feature Envy

The Feature Envy code smell, like the Large Class smell, is almost as likely to occur in core classes as in non-core classes.

Type Checking

The Type Checking code smell was found less than once every 1,000 LOC but was found twice as often in core classes. This was expected, the way Android handles menu item clicks most likely contributed.

Dead Code

The Dead Code code smell is more likely to be found in non-core classes.

Chapter 6

Recommendations

6.1 Introductions

Based on the results in the previous chapter there are a number of recommendations that can be given to Android application developers. Three of these recommendations are selected for a refactoring strategy for code smells. This implementation is done in a Eclipse plugin and the before and after code smell count is given.

For the first case a lot of details are given on how something is done in the plugin, e.g. finding all calls to getView, for the other cases this information is left out.

6.2 Case 1: Adapter Implementation

The Adapter.getView implementation from the Android guidelines has been described in Section 2.2.3.

Improving the getView implementation is done by splitting the method in two parts as discussed before. The first part consists of setting a ViewHolder object, either from a newly created view or from the recycled view. In the case a new view is created, the references of the UI components are set using the getViewById method. The second part consists of setting the views to reflect the data, like a contact name or a contact photo.

6.2.1 Identify

Identifying the getView methods is the first step to changing it. The getView method that needs to be found extends the getView method of android.widget.Adapter.

Once these methods have been found we can look for specific identifiers in the method body to see if it can be improved.

Figure 6.1 shows the code from Section 2.2.3. What we can look for:

- A local field for the ViewHolder which references a class with only fields, no methods or constructors (Line 3).

```
 1 @Override
 2 public View getView(int position, View convertView, ViewGroup
       parent) {

 3   ContactsViewHolder viewHolder;

 4   if (convertView == null) {
 5     convertView = layoutInflater.inflate(R.layout.contact_row,
           parent, false);

 6     viewHolder = new ContactsViewHolder();
 7     viewHolder.txName = (TextView) convertView.findViewById(R.id.
           tvName);
 8     viewHolder.txEmail = (TextView) convertView.findViewById(R.id
           .tvEmail);
 9     viewHolder.txPhone = (TextView) convertView.findViewById(R.id
           .tvPhoneNumber);

10     convertView.setTag(viewHolder);
11   } else {
12     viewHolder = (ContactsViewHolder) convertView.getTag();
13   }

14   Contact contact = getItem(position);
15   viewHolder.txName.setText(contact.getName());
16   viewHolder.txEmail.setText(contact.getEmail());
17   viewHolder.txPhone.setText(contact.getPhoneNumber());

18   return convertView;
19 }

20 class ContactsViewHolder {
21   TextView txName;
22   TextView txEmail;
23   TextView txPhone;
24 }
```

Figure 6.1: Example of a getView method following the Android guidelines.

- An if statement with the left operand being the convertView parameter, the equals or not equals operator and the right operand is null (Line 4).
- Inside the branch of the if/else where the convertView is null there is a inflate method call (Line 5).
- Line 6: A new instance of the ViewHolder class is made.
- Line 7-9: References to the child views will be set to local fields of the ViewHolder class instance just created using calls to findViewById(resourceId).
- Line 10: The ViewHolder instance will be stored inside the view as a tag by calling setTag().
- Line 12: In the branch of the if/else where the convertView is not null, only the ViewHolder instance is retrieved by calling getTag on the convertView.

Line 7 to 9 can be simplified by adding a constructor to the ViewHolder class which takes the view as an argument. In the constructor the findViewById method calls are made.

The second part of the getView method also has certain identifiers:

- Line 14: A call to getItem(position) which gets the model object to show to the user using the view that will be returned.
- Line 15-17: Calls to the different set methods for the child views called on the fields of the ViewHolder class.

This part can be simplified by creating a setItem method in the ViewHolder class that will perform all the calls to the different set methods for the child views. The result can be seen in Figure 6.2.

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

  ContactsViewHolder viewHolder;

  if (convertView == null) {
    convertView = layoutInflater.inflate(R.layout.contact_row, parent, false);

    viewHolder = new ContactsViewHolder(convertView);

    convertView.setTag(viewHolder);
  } else {
    viewHolder = (ContactsViewHolder) convertView.getTag();
  }

  viewHolder.setItem(getItem(position));

  return convertView;
}

class ContactsViewHolder {
  TextView txName;
  TextView txEmails;
  TextView txPhones;

  public ContactsViewHolder(View convertView) {
    txName = (TextView) convertView.findViewById(R.id.tvName);
    txEmails = (TextView) convertView.findViewById(R.id.tvEmails);
    txPhones = (TextView) convertView.findViewById(R.id.tvNumbers);
  }

  public void setItem(Contact contact) {
    txName.setText(contact.getName());
    txEmails.setText(contact.getEmails().toString());
    txPhones.setText(contact.getNumbers().toString());
  }
}
```

Figure 6.2: Refactored Android.getView implementation from Figure 6.1.

Splitting the two parts is done by expanding the use of the ViewHolder class: the findViewById code is moved into the constructor and setting the views to reflect the data is moved into a new method called setItem with the data object as an argument. The number of lines increases but by separating the two functions the getView method code becomes more clear, the possibility of a Long Method code smell in the getView method has decreased.

Performing a refactoring for these changes can be done by an Eclipse plugin. Doing this might not be easy cause of the many different structures the AST can have.

6.2.2 Refactor Implementation

After building an AST we look for the signature of methods that override the method *public abstract android.view.View getView(int, android.view.View, android.view.ViewGroup)* from the class *android.widget.Adapter*.

These are the steps:

- Find the local field that references the ViewHolder class. In the AST of JDeodorant this is done by looking for a LocalVariableDeclarationObject. Since there can be many local fields in the getView method unrelated to the ViewHolder class we look for classes that have no methods or constructors.

- Find the if statement with the convertView being the left operand. We start by getting the name of the second parameter of the getView method, this has to be the left operand. The operator must be equals or not equals. The right operand must be null. We find all calls to findViewById, these will be moved into the constructor of the ViewHolder class.

- Find all method calls performed on the fields in the ViewHolder class. These will be moved into the setItem method of the ViewHolder class.

6.2.3 Limitations

The getView method from Figure 2.3 is the most common one but obviously there can be small changes to the code block while it still functions as expected.

One of the changes the Eclipse plugin doesn't handle is different expressions, e.g. the *convertView == null* expression can also be a *!(convertView instanceof View)* expression. Given this and all the other little changes possible makes it harder to identify certain classes, fields and methods required for the refactoring. For simplicity the Eclipse plugin only works on the most common ViewHolder pattern template. The Eclipse plugin does handle simple changes like a different name for a parameter or local field.

6.3 Case 2: ActivityView Class

The user interface is inflated from a XML into a root View object. Inside this root View object are other views that can be found by calling findViewById(int layoutId). Getting these references is done in the onCreate method.

The references to the Views are usually fields in the Activity class. I suggest making a ActivityViewHolder class to hold the references to the views. This will move a lot of the code from the onCreate method into the new class.

6.3.1 Refactor Implementation

Performing this refactor should not be very difficult:

- Find all Activity.findViewById() calls in the onCreate method.
- All fields that are set from the result of these calls will be moved into the new ActivityViewHolder class and all the references will be updated.
- The calls themselves will be moved into the constructor of this new class.

6.3.2 Eclipse Plugin

The actual Eclipse plugin was not that difficult to make. As said above it finds all calls to findViewById in the onCreate. All the fields that are set using the findViewById call are moved into a inner class of the Activity called ActivityViewHolder. A new field is added to the Activity to refer to ActivityViewHolder. It is set after the setContentView call. Just like with the previous case the fields from the Activity are moved into the ActivityViewHolder class and all the findViewById calls that set these fields are moved into the constructor of the ActivityViewHolder class. Finally the references to the fields are updated to point to the ActivityViewHolder field.

6.4 Case 3: onClick listener in XML

As said above the onCreate method inflates the user interface. It is also used to set all listeners for all the views. This usually means setting a OnClickListener for each view. Because this requires anonymous inner classes this will increase the number of lines by a lot. Android also supports setting a XML attribute named onClick for each view which points to a method that is called when the view is clicked. This simplifies the onCreate method and also improves the readability of the class if a good name is chosen for these methods, something like onClickLoginButton() for when a LoginButton is clicked.

6.4.1 Refactor Implementation

Performing this refactoring might be difficult:

- Find the setContentView(resourceLayoutId) call to get the layout xml file.

- Find all setOnClickListener calls and the view objects it is called on.
- Find the call to findViewById that sets the view object. This findViewById call has the resourceId that corresponds to the view object. From this we can find the xml element in the layout file that corresponds to the view object. We will need to add an 'onClick' attribute to this XML element.
- The method body of the run method in the OnClickListener needs to be moved to a public class method of the activity and the name of that method needs to be set as the value of the 'onClick' attribute.

6.4.2 Eclipse Plugin

It took a while to get it working correctly, mostly because XML editing is not easy to get right in Eclipse.

The Eclipse plugin starts with looking for Activities that have a setContentView(int layoutId) call in their onCreate method. This will tell the layout File we are looking for.

With the correct layout file the plugin looks for all calls to setOnClickListener. There are different AST structures that can point to this call. The expression on which the method is called can be a variable, a method invocation or a parenthesized version of either, for example:

- button1.setOnClickListener(...);
- findViewById(R.id.button1).setOnClickListener(...);
- (findViewById(R.id.button1)).setOnClickListener(...);

The argument of the setOnClickListener call is an anonymous inner class of the type OnClickListener. It has only a single method which is the onClick method. The code block of this method will be used during the refactoring.

From this method call the view id needs to be found. When the expression is a findViewById call it is simple because the argument is the view id but in the case of a variable the place where it is set using the findViewById call needs to be found.

When both the layout file, the setOnClickListener call with the code block and the view id are known the refactoring begins. The onClick attribute is added to the child matching the view id in the layout file. The value of this attribute will be onClick_[ViewId] in the assumption that the developer has chosen a view id that represents the view correctly. Another reason to add the view id to the method name is that it is unique. After saving the layout file the onClick method is added to the Activity class and the code block from the OnClickListener is moved to this method. Finally the setOnClickListener call can be removed.

6.5 Case 4: onClick in MenuItem

The onClick XML attribute is also supported for MenuItem UI components. MenuItems are responsible for the large switch code blocks which result in the Long Method code smells found in the onMenuItemClicked methods.

6.6 Case 5: Resource Code Smells

As described in Section 2.2.1 resources in Android can be referenced in source code using resource ids.

A lot of methods to change properties of views accept both the resource value itself or the resource id. The following two lines of code are exactly the same:

```
imageView.setImageDrawable(getResources().getDrawable(R.
    drawable.image1));

imageView.setImageResource(R.drawable.image1);
```

Obviously the second line of code is easier to read but both are more or less readable. Consider the following case:

```
textView.setCompoundDrawablesWithIntrinsicBounds(
  getResources().getDrawable(R.drawable.image1),
  getResources().getDrawable(R.drawable.image2),
  getResources().getDrawable(R.drawable.image3),
  getResources().getDrawable(R.drawable.image4));

textView.setCompoundDrawablesWithIntrinsicBounds(
  R.drawable.image1, R.drawable.image2,
  R.drawable.image3, R.drawable.image4);
```

Another method that accept both the resource value and the resource id is the TextView.setText method.

Doing a quick search in the apps shows that this is a common error Android developers make. Refactoring the code to the shorter version makes the code a lot easier to read.

6.7 Refactor Results

As explained above an Eclipse plugin was made that performs the refactorings described in the first 3 cases. All 3 refactorings should help limit Long Method code smells.

This section will give the before and after refactoring code smells results by performing the analysis described in 5 again after refactoring.

Because running the refactoring requires a manual check to ensure that everything went ok the results are limited to OO and SipDroid and for finding code smells I will be using both PMD and Checkstyle.

6.7.1 Case 1: Adapter Implementation

The target of case 1 is long getView methods caused by the getView method requiring to do two things. Before refactoring OO had 12 getView methods that were considered Long Method code smells, after refactoring it was reduced to 4. The remaining getView methods did not have the ViewHolder pattern implemented at all which is a requirement for the plugin to function.

6.7.2 Case 2: ActivityView Class

The target of case 2 is long onCreate methods caused by many calls to findViewById to get all references too views. The solution is to introduce a ViewHolder like adapters have to Activities. In OO 13 ActivityViewHolders are added. This did not result in removing any Long Method smells. This means the findViewById calls were not the only reason why some of the onCreate methods were considered Long Method smells. I still believe it is an improvement to have the views in their own class and finding the references in the constructor of that class because it makes the onCreate method easier to understand.

6.7.3 Case 3: onClick listener in XML

The target of case 3 is long onCreate methods caused by many setOnClickListener calls with a large inner anonymous class with the onClick method. Before refactoring OO had 13 onCreate methods that were considered Long Method code smells, after refactoring this is reduced to 7. The remaining onCreate methods still considered Long Method smells are so long because of other listeners with anonymous inner classes. These listeners do not have a corresponding XML attribute so the refactoring used here can not be adapted to also handle these listeners.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

This Section gives an overview of the three research questions (Section 1.4) and where they were answered.

7.1.1 RQ1: What are the main differences between mobile software applications and traditional software applications?

RQ1 was answered in Section 2.2, the differences include:

- Smaller number of developers
- Short lifespan and development cycle
- Regular platform updates
- Limited processing power
- Smaller project size
- External libraries
- Less inheritance
- High Interactive Applications

Limited processing power was further discussed in Section 3.2. In Section 6.2 the adapter implementation, which helps with keeping the required power consumption low, is refactored to make is less susceptible to the Long Method code smell.

7.1.2 RQ2: Which tools can be used to look for code smells in mobile applications and what is the quality of these tools?

RQ2 consists of two parts, which tools can be used to look for code smells and what the quality is of these tools, and both were answered in Section 4.3.

The tools that gave the best results, and therefore were used in the analysis, are JDeodorant, PMD, Checkstyle and UCDetector.

7.1.3 RQ3: Is mobile application code more prone to code smells and if it is how can this risk be limited?

The first part of RQ3, if mobile application code is more prone to code smells, was answered in Section 5.4. The results showed that:

- the Long Method code smell was almost twice as likely to occur in mobile application code.
- the Large Class and Feature Envy code smell were found equally in mobile application code and none mobile application code.
- the Type Checking code smell was twice as likely to occur in mobile application code.
- the Long Parameter List code smell was almost non-existent in mobile application code with less than 1 Long Parameter List code smell per 10,000 LOC. It was found three times as much in none mobile application code but with these low values that might not be very significant.
- the Dead Code code smell was more likely to occur in none mobile application code.

The second part of RQ3, how mobile application code smells can be limited, was answered in Chapter 6. Five refactorings were described and three of them were implemented in a Eclipse plugin to see if the code smells could be reduced. Two of the refactorings were successful with code smells dropping from 12 to 4 for the adapter implementation refactoring and from 13 to 7 for the onClick listener in XML.

7.2 Reliability

The reliability of the answer to the third research question depends on the tools used to find the code smells and the definition of what code is relevant to Android:

- The reliability of the tools is analyzed in Section 4.3.13.

- The reliability of the definition depends on the definition of core classes, the given definition is just one of the possibilities. Independent of Minelli I came to the same definition as he did. Another option could be to define Android relevant code at the method level, e.g. consider all code in a method as Android relevant if it contains at least one call to the Android API.

7.3 Future Work

Below are some of the possibilities to continue this research.

- Finding whether Android code is more prone to any of the other Fowler code smells.
- More metrics to find code smells: Fowler does not specify how finding code smells should be done and points to human intuition as the key to verifying if a code smell needs refactoring. In this paper the most basic metric was used to find the bloater code smells, NLOC. The Long Method for example could be found using a combination of NLOC, Cyclomatic Complexity and Halstead metrics according to Mantyla [13].
- Expanding the Eclipse Refactor Plugin to include the other recommendations.

From a personal point of view I will release the Eclipse Refactor Plugin as open source. I will also try to improve the checks that are done before the refactoring to further limit the possibility that the refactoring results in code that does not compile. I would also like to include the other recommendations and thereby help make the code of Android projects easier to read. I feel this will be helpful given that a lot of developers are starting in mobile application development which means the code they produce while learning might benefit from refactoring tools.

Appendix A

Android Releases

API	Release Name	Release Version(s)	Release Date First Version	No. of Days Since Previous Release
3	Cupcake	1.5	30 April 2009	
4	Donut	1.6	15 September 2009	138
5	Eclair	2.0	26 October 2009	41
6	Eclair	2.0.1	3 December 2009	38
7	Eclair	2.1	21 January 2010	49
8	Froyo	2.2	20 May 2010	119
9	Gingerbread	2.3 - 2.3.2	6 December 2010	200
10	Gingerbread	2.3.3 - 2.3.7	9 February 2011	65
11	Honeycomb	3.0	22 February 2011	13
12	Honeycomb	3.1	10 May 2011	77
13	Honeycomb	3.2	15 July 2011	66
14	Ice Cream Sandwich	4.0 - 4.0.2	19 October 2011	96
15	Ice Cream Sandwich	4.0.3 - 4.0.4	16 December 2011	68
16	Jelly Bean	4.1	9 July 2012	206
17	Jelly Bean	4.2	13 November 2012	127

Table A.1: Release dates of Android versions based on their API number.

Appendix B

Tool Validation

The following files were chosen to use for validation:

- NewsActivity in OO which is the largest Android related source file. The NewsActivity class is 1404 lines of code.
- From SipDroid some of the classes in the org.sipdroid.sipua.ui package were chosen, since this is one of the few packages with Android related code. Most of the SipDroid application contains code to handle the Sip protocol. The following classes I've chosen to analyze:
  - org.sipdroid.sipua.ui.Receiver
  - org.sipdroid.sipua.ui.InCallScreen
  - org.sipdroid.sipua.ui.CallScreen

  The total sum of code lines for these 3 classes is 1638.
- The custom made Android application where code smells were intentionally added.
  - 3 Long Method code smells
  - 1 Large Class code smell
  - 1 God Class code smell
  - 3 Long Parameter List code smells

The code smells that was looked for are Large Class and Long Method. Below is a description of the results for each of the projects.

B.1 App: OO

B.1.1 JDeodorant

God Class

JDeodorant showed three God Class code smells inside NewsActivity, all three were adapters: ColumnAdapter, ArticleAdapter and TwitterAdapter.

- ColumnAdapter is used to provide a ViewPager with the views of each of its 7 pages. JDeodorant suggests 7 class fields that can be extracted along with its methods from the class to make it less of a God Class. Each field references a page in the ViewPager so for all 7 extractions the refactoring can be performed thereby creating classes for each page and removing the code smell from the ColumnAdapter. I agree with this refactoring because it removed the code smell and makes the code easier to understand.
- ArticleAdapter fills a ListView with items, it creates the views and provides access to the data. JDeodorant suggests only one refactoring of one field with one method setting that field. I agree with the refactoring suggested. While looking at the ArticleAdapter I also found two bugs or issues that when solved should simplify the class and make the application faster:
  - The first bug is that the data in the adapter (Articles) is stored twice: once in an ArrayList<Article> in the class itself and once in the superclass. The method to add data to the adapter adds to both of them. It also uses another ArrayList<Integer> to store the ids of each article so no duplicate articles are added. By adding equals(Object) and hashCode() methods to the Article class, the check for duplicates could be performed without using this extra list.
  - The second bug is that in the getView method the ViewHolder pattern is applied but not fully: a new OnClickListener is created each time getView is called.
- TwitterAdapter is almost the same as ArticleAdapter, the same refactoring is suggested and the double holding of data bug is also found here.

Long Method

JDeodorant finds 10 Long Method code smells inside NewsActivity, 8 of the code smells are in adapters and the other 2 are in overridden Activity methods so all are related to Android.

- ColumnAdapter.filterAgendaResults: I agree the method should be refactored but I disagree with the 2 refactorings suggested by JDeodorant. I decided to refactor the code from *if(A) m(); else if(B) m();* to *if(A || B) m();*.
- ColumnAdapter.fillTeamSpinner: I agree the method should be refactored but JDeodorant suggests moving the instantiation of an object and setting 2 attributes of it to a separate method. I would not consider removing these 3 lines of code a fix for the Long Method code smell.
- NewsActivity.onResume: JDeodorant suggests refactoring an if statement with one method invocation in it. I disagree with the refactoring suggested since it's only three lines of code that are easy to understand but it did

help solve a bug: the condition of the if statement compared two Strings based on the address of the object instead of the contents using the equals. This bug may be introduced because the first String has a name teamId, which one would assume is an Integer.

- MatchAdapter.add: This adapter has the same duplicate data bug that all other adapters had. This adapter shows a list of matches. The matches are sorted by date and there are headers in the list showing the date with the relevant matches under it. The problem is the same model object is used (Match) for both the matches and the headers. This means the header needs to override some of the Match methods. JDeodorant suggests doing that outside of the add method, I agree but it would be better if the header would have its own class and thereby removing the need to override which makes the method soo long.

- 3 x Adapter.getView: The next 3 instances are all the getView method of an adapter but JDeodorant does not give a valid refactoring solution. I do agree the methods should be refactored. In each method there are 2 functions performed. One is creating the view and setting the fields of the ViewHolder to the relevant components. The other is setting the values of those components. Instead of doing both in the getView method we can move the 2 functions into the ViewHolder class, which is a logical place for it. This makes the getView method easier to understand and moves changing the values of fields in the ViewHolder class to the ViewHolder.

- ArticleAdapter.loadBanner: The method creates a View that shows a banner image that can be clicked and will open a link. The largest piece of code in this method is used to get the image url and the link url. JDeodorant only shows the method should be refactored by gives no suggestions on how this should be done, my solution would be to make a Utility method out of getting those 2 urls.

- MatchAdapter.fillResultFlags: The method is used to show an image flag based on the 2 letter ID of a country. In the method code is duplicated for both teams of the match. JDeodorant does not give a suggestion but I agree it should be refactored and would remove the duplicate code and just call the method twice, once for each of the two teams

- NewsActivity.onOptionsItemSelected: This method is called when an item in the options menu is selected. The method has one parameter, the MenuItem that was selected. Based on the id of this menuitem an Intent is created and started. JDeodorant suggests doing the creation in a separate method which I agree with, I would also remove the duplicate starting of the intent which is now done separate for each menuitem id.

B.1.2 Checkstyle

Long Method

Checkstyle finds 15 Long Method code smells inside NewsActivity. 8 of them are also found by JDeodorant. Because Checkstyle does not give refactoring suggestions I can only say if I agree with the notion that a given method is a code smell:

- NewsActivity.onOptionsItemSelected: Agree (same as JDeodorant)
- NewsActivity.ColumnAdapter.instantiateItem: Agree, it has a switch for all of the 7 pages, it would be easier to do this in a separate method.
- NewsActivity.ColumnAdapter.alertNoData: Agree, also has a 7 item switch but no contents for each of the switch items so the method can be removed.
- NewsActivity.ColumnAdapter.fillTeamsSpinner: Agree (same as JDeodorant)
- NewsActivity.ColumnAdapter.LoadAgendaArticlesTask.onPostExecute: Agree sort of, not sure how to improve.
- NewsActivity.ColumnAdapter.LoadSupportersArticlesTask.onPostExecute: After looking at this and the other onPostExecute calls for each of the Tasks, it appears they are all alike, it would be easier to make an abstract CommonTask class and let that class do most of the common work.
- NewsActivity.ColumnAdapter.LoadEkArticlesTask.onPostExecute: See above.
- NewsActivity.ColumnAdapter.LoadTeamArticlesTask.onPostExecute: See above.
- NewsActivity.ColumnAdapter.LoadTwitterTask.onPostExecute: See above.
- NewsActivity.ArticleAdapter.getView: Agree (same as JDeodorant)
- NewsActivity.ArticleAdapter.loadBanner: Agree (same as JDeodorant)
- NewsActivity.MatchAdapter.add: Agree (same as JDeodorant)
- NewsActivity.MatchAdapter.getView: Agree (same as JDeodorant)
- NewsActivity.MatchAdapter.fillResultFlags: Agree (same as JDeodorant)
- NewsActivity.TwitterAdapter.getView: Agree (same as JDeodorant)

B.1.3 PMD

Long Method

The results for the Long Method code smell of PMD match those of Checkstyle. This is to be expected given the same parameters given to both tools.

Large Class

PMD reports two Large Class code smells, NewsActivity itself and NewsActivity.ColumnAdapter.

I agree with both code smell reports but was expecting more adapters to be reported as Large Class code smells.

B.2 SipDroid

B.2.1 JDeodorant

God Class

- Receiver: JDeodorant correctly suggests refactoring this class but only wants to move the progress method and onState method, I would add the onText method and other static methods that do not belong in a BroadcastReceiver, which only function is to implement the onReceive method. Most of the other methods can be moved to Utility classes and helper classes.

- InCallScreen: As with the Receiver class, JDeodorant suggests refactoring this class only in a limited way, it wants to split one method and the corresponding fields. I agree with the refactoring suggestion made, the result after refactoring looks more clear and the functionality placed in the new class did not fit well with the rest of the class.

- CallScreen: As with the above classes, only one item is suggested for refactoring and like InCallScreen I agree with result but to a lesser degree, the extracted class has less functionality compared to the extracted class created in InCallScreen.

Long Method

- Receiver.onReceive: This is the only method required in the Receiver class, this one has to be overridden for the Receiver to correctly implement BroadcastReceiver. The method is 128 lines long and JDeodorant suggests 5 different refactor options. After performing each suggested refactoring the method looks more like how I would have refactored it myself. Not every refactoring was done without causing compile errors but these were easy to solve.

- Receiver.onText: This is a static method that is used to update the text of the notification in the system bar, e.g. a missed call warning. I agree that the method should be refactored.

- Receiver.onState: Two refactorings are suggested for this method. I agree with neither of them but I do think the method should be refactored.

- CallScreen.opPause: The onPause method is 7 lines in size, 4 of them are used to close the socket like this:

```
        if (socket != null) {
            socket.close();
            socket = null;
        }
```

I agree that it would look better if there was a closeSocket method that was called from the onPause method, which is what JDeodorant suggests.

- CallScreen.onOptionsItemSelected: This is an Android method that is called when a menuitem is selected. A better option would be to use the setOnMenuItemClickListener method on each MenuItem.

- InCallScreen.onResume: I agree this method should be refactored. JDeodorant gives two possible refactoring options, both of them related to one line of code.

- InCallScreen.onPause: I agree this method should be refactored. It appears to do 3 or 4 different things, this can be simplified into separate methods.

- InCallScreen.answer: I agree this method should be refactored, it is short but it may be unclear what is done inside the method without refactoring. I do not agree with the refactoring options that JDeodorant gives.

- InCallScreen.reject: This method appears to be the opposite of the tanswer method, not only in functionality but also in the method body. The same comments given to the answer method apply to this method.

B.2.2 Checkstyle

Long Method

- CallScreen.onResume().new Thread().run: I agree this method should be refactored, the entire anonymous inner class could be moved into a separate class.

- CallScreen.onOptionsItemSelected: Same as JDeodorant

- CallScreen.onResume: Same as JDeodorant

- InCallScreen.new Handler().handleMessage: I agree this method should be refactored, the Handler class can be moved into a separate class but I think splitting up the method would be a better choice for Handlers.

- InCallScreen.onResume().new Thread().run: I agree this method should be refactored, the entire anonymous inner class could be moved into a separate class to limit the length of the onResume method.

- InCallScreen.initInCallScreen: I agree this method should be refactored, it can be simplified by moving two parts of code that fill a Map.

- InCallScreen.onKeyDown: This method is called when the user pressed the menu button, the call button, the back button, the camera button, the volume up and down button. The type of button is identified by a switch statement. Inside this switch statement the refactoring can be to call a method for each of the different keys which will simplify the onKeyDown method.

- Receiver.onReceive: Same as JDeodorant

- Receiver.onState: Same as JDeodorant

- Receiver.onText: Same as JDeodorant

B.2.3 PMD

Long Method

As with OO, the results of PMD Long Method match those of CheckStyle LongMethod.

Large Class

PMD finds three LargeClass code smells, the same as JDeodorant did.

B.3 Own Test Application

The custom test application has 1 God Class code smell which also is a Large Class smell, 3 Long Method smells and 3 Long Parameter List smells.

B.3.1 JDeodorant

JDeodorant found the God Class code smell and all 3 Long Method code smells.

As said before, JDeodorant is able to identify God Class code smells and not like the other tools Large Class code smells. There is a relation between the two code smells since in a lot of cases where one is found the other can also be found but there is a difference between the two which can be seen by stripping a class down to what JDeodorant still considers a God Class.

```
public class GodClass {

    private String str1;
    private String str2;

    private void loadString1() {
        str1 = "abc";
    }

    private void loadString2() {
        str2 = "abc";
    }
}
```

B.3.2 Checkstyle

Checkstyle found the 3 Long Method code smells and also all 3 Long Parameter List code smells.

B.3.3 PMD

PMD is using the same parameters for the code smells so like Checkstyle, it also found the 3 Long Method code smells and also all 3 Long Parameter List code smells. It also found the large class just like JDeodorant found the God Class.

Appendix C

Raw Analysis Results

C.1 Lint

Description	**OO**	**SipDroid**
Looks for unused resources	304	74
Checks for incomplete translations where not all strings are translated	0	148
Looks for uses of "dp" instead of "sp" dimensions for text sizes	89	3
Looks for overdraw issues (where a view is painted only to be fully painted over)	60	3
Looks for use of the "px" dimension	0	41
Finds calls to locale-ambiguous String manipulation methods	3	32
Checks for openFileOutput() and getSharedPreferences() calls passing MODE_WORLD_WRITEABLE	26	0
Ensures that image widgets provide a contentDescription	3	21
Finds API accesses to APIs that are not supported in all targeted API versions	0	20
Looks for hardcoded text attributes which should be converted to resource lookup	6	7
Looks for ellipsis strings (...) which can be replaced with an ellipsis character	0	13
Looks for miscellaneous typographical problems like replacing (c) with ©	0	13
Checks for duplicate ids across layouts that are combined with include tags	9	1
Checks that files with DOS line endings are consistent	8	0
Ensures that Handler classes do not hold on to a reference to an outer class	3	5
Looks for usages of "new" for wrapper classes which should use "valueOf" instead	6	0
Looks for invocations of android.webkit.WebSettings.setJavaScriptEnabled	5	0
Checks whether a parent layout can be removed.	2	3
Looks for inefficient weight declarations in LinearLayouts	1	4
Using SimpleDateFormat directly without an explicit locale	4	0
Looks for typos in messages	2	2
Looks for text fields missing inputType or hint settings	0	4
Looks for problems with wakelock usage	0	4
Looks for hardcoded references to /sdcard	0	3
Checks for exported activities that do not require permissions	0	3
Looks for usages of deprecated layouts, attributes, and so on.	0	3
Looks for code creating a Toast but forgetting to call show() on it	2	0
Ensures that icons provide custom versions for all supported densities	2	0
Checks that the manifest specifies a targetSdkVersion that is recent	1	1
Ensures that Activities, Services and Content Providers are registered in the manifest	1	1
Ensure that allowBackup is explicitly set in the application's manifest	1	1
Looks for opportunities to replace HashMaps with the more efficient SparseArray	0	2
Looks for LinearLayouts which should set android:baselineAligned=false	0	2
Checks whether a root <FrameLayout> can be replaced with a <merge> tag	1	0
Checks for manifest problems like <uses-sdk> after the <application> tag	0	1
Looks for layout params that are not valid for the given parent layout	0	1
Checks for exported receivers that do not require permissions	0	1

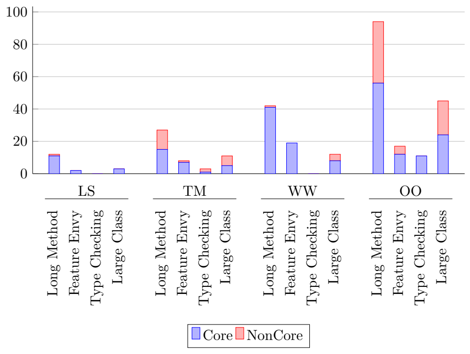
Table C.1: Lint results of OO and SipDroid

C.2 JDeodorant

C.2.1 Commercial Applications

	Large Class	Long Method	Feature Envy	Type Checking
OO	24 / 45	56 / 94	12 / 17	11 / 11
LS	3 / 3	11 / 12	2 / 2	0 / 0
TM	5 / 11	15 / 27	7 / 8	1 / 3
WW	8 / 12	41 / 42	19 / 19	0 / 0
Total	40 / 71	123 / 175	40 / 46	12 / 14

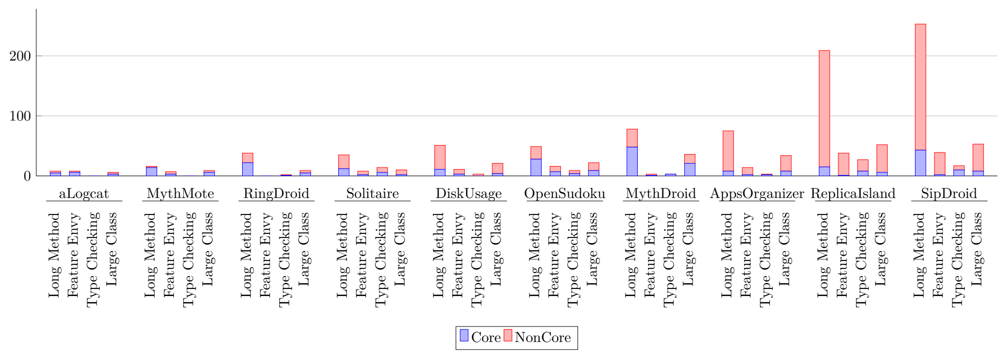
Table contains number code smells found in core classes and total number of code smells found.



C.2.2 Open Source Applications

	Large Class	Long Method	Feature Envy	Type Checking
SipDroid	8 / 53	43 / 253	2 / 39	10 / 17
aLogcat	3 / 6	5 / 8	6 / 8	0 / 0
AppsOrganizer	8 / 34	8 / 75	2 / 14	2 / 3
DiskUsage	4 / 21	11 / 51	3 / 11	0 / 3
MythDroid	21 / 36	48 / 78	1 / 3	3 / 3
MythMote	6 / 9	14 / 16	3 / 7	0 / 0
OpenSudoku	9 / 22	28 / 49	7 / 16	4 / 9
ReplicaIsland	6 / 52	15 / 209	1 / 38	8 / 27
RingDroid	5 / 9	22 / 38	0 / 0	1 / 2
Solitaire	2 / 10	12 / 35	2 / 8	6 / 14
Total	72 / 252	206 / 812	27 / 144	34 / 78

Table contains number code smells found in core classes and total number of code smells found.

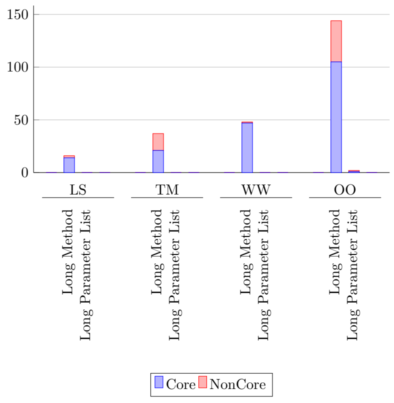


C.3 Checkstyle

C.3.1 Commercial Applications

	Long Method	Long Parameter List
OO	105 / 144	1 / 2
LS	14 / 16	0 / 0
TM	21 / 37	0 / 0
WW	47 / 48	0 / 0
Total	187 / 245	1 / 2

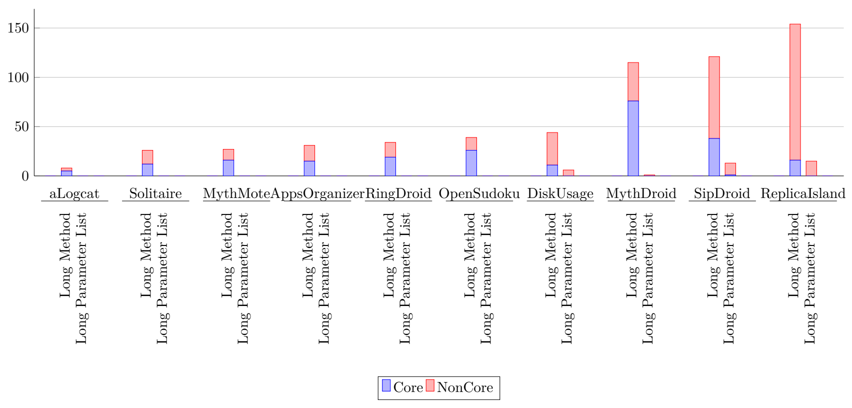
Table contains number code smells found in core classes and total number of code smells found.



C.3.2 Open Source Applications

	Long Method	Long Parameter List
SipDroid	38 / 121	1 / 13
aLogcat	5 / 8	0 / 0
AppsOrganizer	15 / 31	0 / 0
DiskUsage	11 / 44	0 / 6
MythDroid	76 / 115	0 / 1
MythMote	16 / 27	0 / 0
OpenSudoku	26 / 39	0 / 0
ReplicaIsland	16 / 154	0 / 15
RingDroid	19 / 34	0 / 0
Solitaire	12 / 26	0 / 0
Total	234 / 599	1 / 35

Table contains number code smells found in core classes and total number of code smells found.

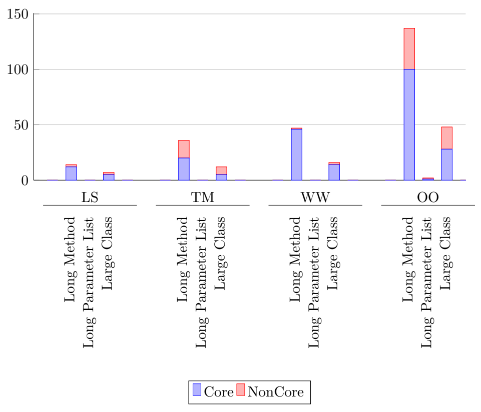


C.4 PMD

C.4.1 Commercial Applications

	Large Class	Long Method	Long Param List
OO	28 / 48	100 / 137	1 / 2
LS	5 / 7	12 / 14	0 / 0
TM	5 / 12	20 / 36	0 / 0
WW	14 / 16	46 / 47	0 / 0
Total	52 / 83	178 / 234	1 / 2

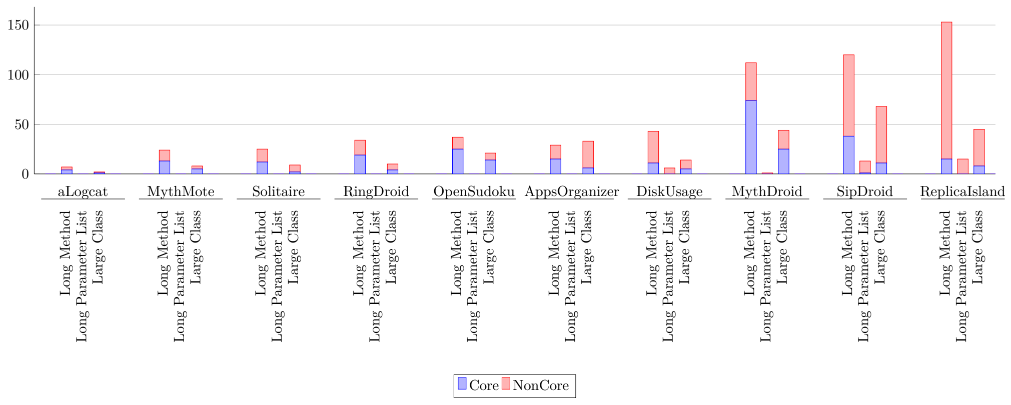
Table contains number code smells found in core classes and total number of code smells found.



C.4.2 Open Source Applications

	Large Class	Long Method	Long Param List
SipDroid	11 / 68	38 / 120	1 / 13
aLogcat	1 / 2	4 / 7	0 / 0
AppsOrganizer	6 / 33	15 / 29	0 / 0
DiskUsage	5 / 14	11 / 43	0 / 6
MythDroid	25 / 44	74 / 112	0 / 1
MythMote	5 / 8	13 / 24	0 / 0
OpenSudoku	14 / 21	25 / 37	0 / 0
ReplicaIsland	8 / 45	15 / 153	0 / 15
RingDroid	4 / 10	19 / 34	0 / 0
Solitaire	2 / 9	12 / 25	0 / 0
Total	81 / 254	226 / 584	1 / 35

Table contains number code smells found in core classes and total number of code smells found.

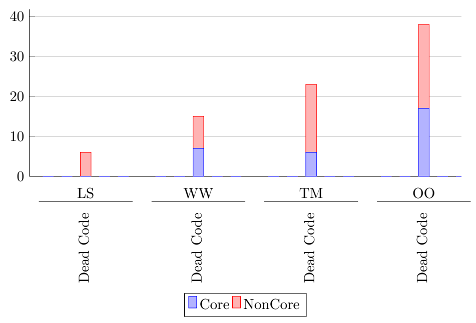


C.5 UCDetector

C.5.1 Commercial Applications

	Dead Code
OO	17 / 38
LS	0 / 6
TM	6 / 23
WW	7 / 15
Total	30 / 82

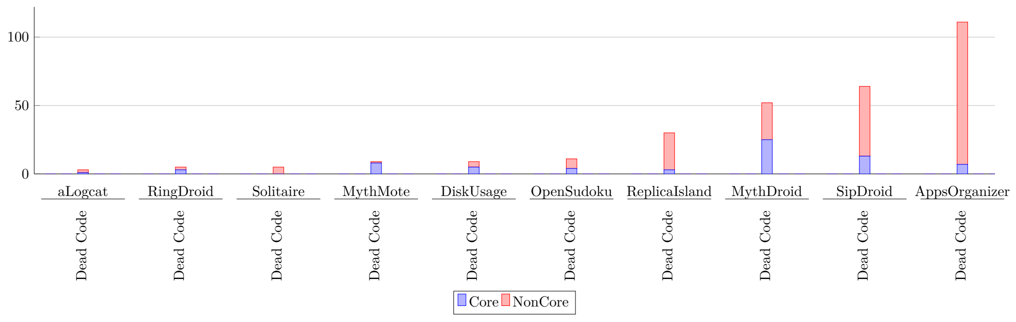
Table contains number code smells found in core classes and total number of code smells found.



C.5.2 Open Source Applications

	Dead Code
SipDroid	13 / 64
aLogcat	1 / 3
AppsOrganizer	7 / 111
DiskUsage	5 / 9
MythDroid	25 / 52
MythMote	8 / 9
OpenSudoku	4 / 11
ReplicaIsland	3 / 30
RingDroid	3 / 5
Solitaire	0 / 5
Total	69 / 299

Table contains number code smells found in core classes and total number of code smells found.



Bibliography

[1] Android api - intent. `http://developer.android.com/reference/android/content/Intent.html`; accessed 01-Mrt-2013.

[2] Checkstyle website. `http://checkstyle.sourceforge.net/`; accessed 05-Feb-2013.

[3] Internet survey on mobile app development time. `http://www.kinvey.com/blog/2086/how-long-does-it-take-to-build-a-mobile-app`; accessed 17-Jan-2013.

[4] Lint website. `http://tools.android.com/tips/lint`; accessed 05-Feb-2013.

[5] Pmd website. `http://pmd.sourceforge.net/`; accessed 05-Feb-2013.

[6] Pmd wikipedia page. `http://en.wikipedia.org/wiki/PMD_(software)`; accessed 15-Aug-2013.

[7] Ucdetector website. `http://www.ucdetector.org/`; accessed 06-Feb-2013.

[8] F Arcelli Fontana, P Braione, and M Zanoni. Automatic detection of bad smells in code: An experimental assessment. *The Journal of Object Technology*, 11(2):1 – 38, 08 2012.

[9] Lerina Aversano, Luigi Cerulo, and Massimiliano Di Penta. How clones are maintained: An empirical study. In *In Proceedings of the 11th European Conference on Software Maintenance and Reengineering (CSMR'07*, pages 81–90, 2007.

[10] J. Daly, A. Brooks, J. Miller, M. Roper, and M. Wood. The effect of inheritance on the maintainability of object-oriented software: an empirical study. In *Proceedings of the International Conference on Software Maintenance*, ICSM '95, pages 20–, Washington, DC, USA, 1995. IEEE Computer Society.

[11] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA, 1999.

[12] Marion Gottschalk, Mirco Josefiok, Jan Jelschen, and Andreas Winter. Removing energy code smells with reengineering services. In Ursula Goltz, Marcus Magnor, Hans-Jürgen Appelrath, Herbert K. Matthies, Wolf-Tilo Balke, and Lars Wolf, editors, *Beitragsband der 42. Jahrestagung der Gesellschaft für Informatik e. V. (GI)*, volume 208, pages 441–455. Bonner Köllen Verlag, 2012.

[13] Mika Mäntylä, Jari Vanhanen, and Casper Lassenius. A taxonomy and an initial empirical study of bad smells in code. In *Proceedings of the International Conference on Software Maintenance*, ICSM '03, pages 381–, Washington, DC, USA, 2003. IEEE Computer Society.

[14] C. Marinescu, R. Marinescu, P. F. Mihancea, and R. Wettel. iplasma: An integrated platform for quality assessment of object-oriented design. In *In ICSM (Industrial and Tool Volume*, pages 77–80. Society Press, 2005.

[15] Roberto Minelli and Michele Lanza. Software analytics for mobile applications, insights & lessons learned. In *Proceedings of the 2013 17th European Conference on Software Maintenance and Reengineering*, 2013.

[16] N. Moha, Y.-G. Gueheneuc, L. Duchien, and A.-F. Le Meur. Decor: A method for the specification and detection of code and design smells. *Software Engineering, IEEE Transactions on*, 36(1):20 –36, jan.-feb. 2010.

[17] Emerson Murphy-Hill and Andrew P. Black. An interactive ambient visualization for code smells. In *Proceedings of the 5th international symposium on Software visualization*, SOFTVIS '10, pages 5–14, New York, NY, USA, 2010. ACM.

[18] N. Tsantalis. Evaluation and improvement of software architecture: Identification of design problems in object-oriented systems and resolution through refactorings. 2010.