# Quality Smells

für

Android-Applikationen

Komplexpraktikum Sommersemester 2014

TU-Dresden, Lehrstuhl Softwaretechnologie

Betreuer:

Jan Reimann

Bearbeitende Studenten:

Ronny Marx, Peter Höhne

Abbildung 1: http://www.androidgirl.ca

*Gliederung*

1. Aufgabenstellung
2. Motivation
3. Analyse der Smells
   - 3.1. Slow Loop
   - 3.2. Uncached View
   - 3.3. Unclosed Closable
   - 3.4. Unnecessary Permission
   - 3.5. Untouchable
4. Fazit

## *1. Aufgabenstellung*

In diesem Komplexpraktikum sollen die in unserem Katalog enthaltenen Quality Smells umgesetzt werden. Dazu gehört die Definition der Struktur eines Smells mit Hilfe des Werkzeugs EMF-IncQuery, sowie die Implementierung zugehöriger Refactorings mittels Refactory.

## *2. Motivation*

In Software-Entwicklungsprozessen werden neben funktionalen Anforderungen auch nichtfunktionale Anforderungen an die Qualitätseigenschaften der zu entwickelnden Anwendung spezifiziert. Derartige Qualitätsanforderungen spielen gerade bei mobilen Anwendungen (wie für das Android-System) eine wichtige Rolle. So kann bspw. festgelegt werden, dass der Energieverbrauch einer Anwendung gering zu halten ist. In einer vorangegangenen studentischen Arbeit wurde ein Katalog erarbeitet, der eine Liste sogenannter Quality Smells für Android-Anwendungen enthält. Ein Quality Smell beschreibt dabei die Struktur eines bestimmten Szenarios, das sich bei Vorhandensein im Code negativ auf die Qualitätseigenschaften auswirkt. Außerdem wird angegeben, welche Umstrukturierungen durchgeführt werden können, um den Quality Smell zu beseitigen, dabei aber das implementierte Verhalten nicht zu verändern. Dieses Vorgehen nennt man qualitäts-bewusstes Refactoring, wofür am Lehrstuhl das Tool Refactory entwickelt wurde.

# 3. Analyse der Smells

## 3.1 Slow Loop

***Context***
Implementation

**Affects**
Efficiency

**Problem**
A slow version of a for-loop is used.

**Refactorings**
Enhanced For-Loop

### *Beschreibung*

Wird über ein Object iteriert, welches vom Typ „Iterable“ erbt, so ist es oft günstiger, eine erweiterte for-Schleife für diese Iteration zu verwenden. Die Ursache dafür ist, dass im Allgemeinen angenommen werden kann, dass der Iterator, welcher bei der erweiterten for-Schleife zum Einsatz kommt, effizienter ist als der indexbasierte Zugriff auf Elemente des iterierbaren Typs.

### *Annahmen*

Die Schleife ist von der Form

```
for (INDEX = 0; BEDINGUNG; INKREMENT){
        //...
        FELDZUGRIFF
        //...
}
```

| | | |
|---|---|---|
| INDEX | beliebig | Eine Variable, welche für den Indexzugriff verwendet wird (muss in der Schleife mit 0 initialisiert werden) |
| BEDINGUNG | INDEX < FELDVARIABLE.size()<br>INDEX <= FELDVARIABLE.size() -1 | Schleifenabbruchbedingung |
| FELDVARIABLE | beliebig | Die Variable, welche iterierbar ist (muss Iterable erweitern) |
| INKREMENT | INDEX++<br>++INDEX | |
| FELDZUGRIFF | FELDVARIABLE.get(INDEX) | Andere Feldzugriffe werden nicht unterstützt. Der Feldzugriff kann sich an einer beliebigen Stelle im |

| | | |
|---|---|---|
| | | Ausdruck befinden. Findet sich ein andersartiger zugriff auf die Variable, so kann kein Refactoring angeboten werden. |

***Refactoring:***

Das Refactoring umfasst folgende Abbildungen:

Sei die Feldvariable in folgender Form deklariert:
FeldTyp<ElementTyp> FELDVARIABLE = …

So wird der Schleifenkopf wie folgt ersetzt:
for (INDEX = 0; BEDINGUNG; INKREMENT)
→for (Elementtyp elementRefactored : FELDVARIABLE)

Im Schleifenkörper werden die Vorkommen von Feldzugriffen ersetzt:
FELDVARIABLE.get(i) → elementRefactored

***Zukünftige Arbeiten:***

- *Prüfung, ob die Indexvariable im Schleifenkörper vorkommt.Wenn ja, kann entweder kein Refactoring angeboten werden oder Deklaration und Inkrement müssen im Schleifenkörper angegeben werden.*
- *weitere Bedingungen könnten in betracht gezogen werden*
- *das Inkrement INDEX = INDEX + 1 könnte noch mit in Betracht gezogen werden.*

## 3.2 Uncached View

**Context**
UI

**Affects**
Efficiency

**Problem**
Scrolling of ListViews or switching between pages of ViewPager could be slow.

**Refactorings**
View Holder

### *Beschreibung*

Häufige Zugriffe auf die Methode findViewById() können sich negativ auf die Performance auswirken. Dies tritt Beispielsweise beim Scrollen durch Listen auf.

### *Annahmen*

Die Häufigkeit der Zugriffe kann nicht ohne Weiteres entschieden werden. So kann es bei Callbacks zu häufigen Aufrufen kommen. Somit wird der Einfachheit halber angenommen, dass jeder Aufruf der Methode findViewById() potentiell häufig vorkommen kann und stellt somit einen Smell dar.

### *Refactoring*

In der aufrufenden Klasse wird eine neue private Klasse eingeführt:

```
private static class FindViewCache{
      private static SparseArray<View> cache =  new SparseArray<View>();

      public static View getCachedView(Activity activity, int viewID){
            View result = cache.get(viewID);

            if(result==null){
                  result = activity.findViewById(viewID);
                  cache.put(viewID, result);
            }

            return result;
      }
}
```

Weiterhin werden alle Zugriffe auf findViewById wie folgt ersetzt:

- findViewById(VIEWID) →FindViewCache.getCachedView(this, VIEWID)
- this.findViewById(VIEWID) →FindViewCache.getCachedView(this, VIEWID)

***Zukünftige Arbeiten:***

- Eventuell Auslagerung in eine nicht eingebettete Klasse

## *3.3 Unclosed Closable*

**Context**
Implementation

**Affects**
Memory Efficiency

**Problem**
An object implementing the java.io.Closeable is not closed

**Refactorings**
Close Closeable

### *Beschreibung*

Ein Objekt, das die Klasse *Closeable* nutzt ist eine Quelle oder ein Ziel von Daten. Damit die Daten, welche von diesem Objekt genutzt werden wieder freigegeben werden können muss das Objekt mit *close()* geschlossen werden.

### *Annahmen*

Im Pattern wird dabei geprüft ob ein Interface vom Typ *Closeable* existiert und ob dieses per Methodenaufruf *close()* wieder geschlossen wird.
Mehrmaliges aufrufen von *close()* hat dabei keinen weiteren Effekt.

### *Refactoring*

Beim Aufruf von Closeable wird geprüft ob ein zum Interface gehörender Aufruf von close() existiert.

```
      neg find findClassParam(s);
      OrdinaryParameter.name(t,s);
      find findClass(c,t);
```

Das Pattern liefert dabei Klassenname c und Parameter t an das Refactoring.
Im Refactoring wird nach der Klasse c das close() mit dem Parameter eingefügt

```
Statement  me = (Statement) parsePartialFragment(
                          t.toString()+CLOSE,
                          MembersPackage.Literals.CLASS_METHOD,
                          ClassMethod.class).get(0);

            c.addAfterContainingStatement(me);
```

### *Zukünftige Arbeiten*

Nutzung von AutoCloseable.

## *3.4 Unnecessary Permission*

> **Context**
> Implementation
>
> **Affects**
> Security,User Conformity,User Experience
>
> **Problem**
> The app requests several permissions, some of them are not needed, as they can be replaced easily.
> The more permissions an app needs, the more suspicious it is for the user.
>
> **Refactorings**
> Use activity intent

### *Beschreibung*

Fordert eine App mehr Zugriffsbefugnisse als sie eigentlich benötigt, soll dies dem Entwickler angezeigt werden.

### *Annahmen*

Alle Befugnisse werden in der MANIFEST.xml definiert. Da ein Zugriff auf XML-Dateien per Query nicht implementiert ist, kann nur der Aufruf eines Intents y,als Nutzung einer Befugniss genutzt werden um den Entwickler darauf hinzuweisen die Befugniss zu überprüfen.

```
pattern useIntent(y,z) ={
      NewConstructorCall.typeReference(f,g);
      NamespaceClassifierReference.classifierReferences(g,c);
      ClassifierReference.target.name(c, "Intent");
      NewConstructorCall.arguments(f,x);
      IdentifierReference.next(x,z);
      IdentifierReference.target.name(z,y);
      }
```

### *Refactoring*

Ein Refactoring ist bei diesem Smell derzeit nicht möglich.
Der Benutzer wird über die Intents informiert und gefragt, ob er diese benötigt. Dabei wird je nach Intent eine angepasste Meldung ausgegeben (Kamera, SMS, Kontaktliste).

### *Zukünftige Arbeiten*

Einbindung des XML-Metamodells zur Abfrage der MANIFEST.XML, um Permissions mit dem Code abzugleichen.

## *3.5 Untouchable*

> **Context**
> UI
>
> **Affects**
> User Experience,Accessibility
>
> **Problem**
> If a touchable ui elements size is less then 48dp (ca. 9mm).
> See Accessibility: Are You Serving All Your Users?
>
> **Refactorings**
> Show small elements.

### *Beschreibung*

Bei diesem Smell wird nach Elementen deren Höhe oder Breite kleiner als 48dp ist gesucht.

### *Annahmen*

Dabei wird im Pattern nach den LayoutParams des genutzten Layouts gesucht und die Größe des jeweiligen Elementes geprüft. Dabei wird die IdentifierReference genutzt. Wenn in den LayoutParams Integerwerte zur Angabe der Größe genutzt werden kann per check() deren Größe mit den Anforderungen (48dp) verglichen werden. Da eher selten die Größe von Elementen direkt im Javacode angegeben werden und oft FILL_PARENT (deprecated), WRAP_CONTENT und MATCH_PARENT genutzt werden, wobei die Größe an die Elternelemente angepasst wird, ist ein Hinweis die Größe in der XML-Datei zu prüfen angebracht.

### *Refactoring*

Das Refactoring passt Elemente, welche den Anforderungen nicht entsprechen, an. Dabei wird die Liste der Dezimalzahlen geprüft und entsprechend zu geringe Werte angepasst.

### *Zukünftige Arbeiten*

Die Größe von 48dp wird als zu groß erachtet, auch Elemente mit deutlich geringerer Größe (z.B. 30dp) sind i.d.R. noch gut bedienbar. Eine Einbindung des XML-Metamodells zur Überprüfung der Elementgröße im XML-View ist erfolgversprechender, da oft die Werte nur dort angegeben werden.

## *4. Fazit*

Der Zugriff auf die XML-Dateien würde die Nutzbarkeit einiger Smells deutlich erhöhen, da im WYSIWYG-Editor in der Androidentwicklung zumeist XML-Dateien genutzt werden.