Specification of Android Bad Smells

1 Leaking Inner Class

During the development of Android applications, developers define non-static inner classes. If a non-static inner class holds a reference of outer class, then it could be a memory leak.

Context: Implementation
Affects: Memory Efficiency
Problem: Non-static inner classes holding references to its outer class. This could lead to a memory leak.
Refactoring: Instead of non-static, introduce static classes.
Resolve: Performance and Memory Efficiency.
Detection Rule:

(Class A is a concrete class) And (Class B is a non-static inner class) And (Class B has a reference of Class A)

2 Member-Ignoring Method

The intent of a method in a class is to perform operations on attributes of that class. A method that has nothing to do with attributes of a class is violation of principle of OO programming.

Context: Implementation

Affects: Efficiency
Problem: Non-static methods that don't use any fields should not be in that particular class.
Refactoring: Introduce that static method in any other utility class.
Resolve: Efficiency
Affects: Performance and efficiency
Detection Rule:

(Class A is a concrete class) And (M is the method in class A) And (M is not accessing any attribute of class A)

3 No Low Memory Resolver

Usually, mobile systems are assembled with small RAM and no Virtual memory for swap space to free memory. In that case, Android OS provides a mechanism to help system manger in the context of memory. The override-able predefined method in Activity class “onLowMemory” is called when there is a need to free all background processes.

Context: Implementation
Affects: Memory Efficiency, User Experience and Stability
Refactoring: Override onLowMemory in all Activities of Application.
Resolve: Memory Efficiency, Stability, User Experience
Affects: Performance, Efficiency
Detection Rule:

(Class A is a concrete class extending by Activity class) And (There is no overridden method name as onLowMemory)

4 Internal Getter/Setter

In Android mobile development, accessing fields using getter/setter internally is expensive. Accessing a field directly is three time faster than using getter and setter method. There are also recommendations that it is a best practice to use getters and setters instead of direct access as it is a conventional and the code is more maintainable.

Context: Implementation
Problem: Accessing internal fields by setters and getters in Android is an expensive transaction.
Affects: Performance and efficiency.
Refactoring: Direct field access instead of using set and get methods and only use getters and setters in public Application Programming Interface (API).
Resolve: Efficiency and Performance
Detection Rule:

(Class A is a concrete class) And (In the body of class A there is call to its internal set or get methods)

5 Inefficient Data Format and Parser

Tree parsers similar to XML parsers are too expensive and complex for Android mobile systems. They reduce the device performance requiring extensive memory and processing power.

Context: Network, IO
Affects: Performance, Efficiency
Problem: The use of TreeParsers is too slow.
Refactoring: Use efficient data parser and format.
Detection Rule:

(Class A is a concrete class) And (DocumentBuilderFactory, DocumentBuilder, NodeList are built-in java classes) And (Class A is using any method from these classes)

6 Inefficient Data Structure

Data structures are tools to facilitate programmers to manage data efficiently but improper usage of data structures is expensive and not handy for the development of mobile applications. Hashmap is an efficient data structure that provides a one to one mapping as a key value pair. Using hashmap as Integer to object mapping is expensive and slow.

Context: Implementation
Affects: Performance and Efficiency
Problem: The mapping of an integer to an object as, using HashMap<Integer, Object> is slow.
Refactoring: Use Efficient Data Structure. An alternative to Hashmap is SparseArray which is less expensive comparatively.
Resolves: Performance, Efficiency
Detection Rule:

(Class A is a concrete class) And (Class A is using Hashmap as Integer as a key and any object as representation of value)

7 Leaking Thread

Threads are used to share responsibilities. In Android mobile applications, the default thread starts as the application runs named as User Interface (UI) or main thread. Using threads other than UI or main thread in application breaks down the target work by doing extensive calls in the background like Network calls. Starting a new thread and not discontinuing it after completion of a task is wastage of resources.
Context: Implementation
Affects: Memory Efficiency
Problem: Leaking memory.
Refactoring: Introduce Run Check Variable or use Asyntask (a built-in thread handler for Android application)
Resolves: Memory Efficiency and performance
Detection Rule:

(Class A is a concrete class) And (Class A is using Thread for background tasks) And (Thread is started and there is no statement to stop a thread)

8 Slow For Loop

In Android mobile applications, traditional for loop used for iterations is slow and reduces the efficiency of applications. Instead of using traditional for loop, an enhanced version of for loop is provided by Java-5 that should be used.
Context: Implementation
Affects: Efficiency
Problem: A traditional slow version of a for-loop is used for iteration.
Refactoring: Use an enhanced For-Loop instead of traditional for loop.
Resolves: Efficiency
Detection Rule:

(Class A is a concrete class) And (Class A is using traditional for loop F for iteration) And (For loop F is bound by initialization, condition check and increment or decrement)

9 Public Data

Android system provides shared preferences in order to save data for an application in the form of key value pairs. There are two modes for accessing shared preferences (public and private). Using public mode of shared preferences like MODE_WORLD_READABLE or MODE_WORLD_WRITEABLE is a security risk as the application's private data is available publically.
Context: Implementation
Affects: Security, User Conformity, User Experience

Problem: Private data is kept in a store that is publicly accessible (by other applications).
Refactoring: Set private mode as Context.MODE_PRIVATE
Resolves: User Conformity, Security
Detection Rule:

(Class A is a concrete class) And (Class is using shared preferences) And (Shared preferences are accessed without private mode)

10 Durable WakeLock

In mobile applications, a wake lock is required to keep the device in opening mode by using CPU, Networks and Sensors etc. It is recommended that all the resources of system should be freed after completion of a task. Acquiring a wake lock without any time frame is waste of resources especially for battery energy.
Context: Implementation, UI
Affects: Energy Efficiency
Problem: inefficient use of resources.
Refactoring; Acquire WakeLock with timeout
Resolves: Energy Efficiency
Detection Rule:

(Class A is a concrete class) And (Class is accessing WakeLock to wake an Android device) And (Acquired WakeLock is not based on a timeout)

11 Rigid Alarm Manager

Android mobile systems provide alarm services to their users in order to trigger event-based wake-ups. It may be possible that there are several operations being executed in the sense of system's alarms. It is quite helpful if all the alarms bundled together to improve efficiency and performance of system ensure optimum consumption of energy.
Context: Implementation
Affects: Efficiency, Energy Efficiency
Problem: Using different and independent alarm services is more energy consuming.
Refactoring: The latest recommendation about usage of Alaram services is to use AlarmManager.setInexactRepeating() method instead of AlarmManager.set() or AlarmManager.setRepeating().
Resolves: Performance, Energy Efficiency
Detection Rule:

(Class A is a concrete class) And (Class A is using Alarm Services) And (AlarmManager is calling set or setRepeating method)

12 Unclosed Closable

In java, there is a Closeable interface used to close or release resources after the job is completed. The class implementing that interface is supposed to call a closed method defined in Closable interface. If there is a presence of Closeable Implementation but absence of a call to close method then it is an anti-pattern.
Context: Implementation
Affects: Memory Efficiency
Problem: An object implementing the java.io.Closeable is not closed
Refactoring: Close Closable
Resolves: Optimization of resources and Memory Efficiency
Detection Rule:

(Class A is a concrete class) And (Class A is implementing a Closable interface from java library) And (Class A is not calling close method of Closeable)

13 Debuggable Release

In Android application development, an attribute android:debuggable is defined with value true in order to debug the application. But in release mode, its value should be changed to false as it is a serious security threat.

Context: Implementation
Affects: Security
Problem: The attribute android:debuggable with value true in Manifest file.
Refactoring: Remove the android:debuggable attribute or set it to false.
Resolves: Security issue.
Detection Rule:

(Search AndroidManifest.xml file in Android project) AND (Check the value of attribute android:debuggable)

14 Dropped Data

In Android applications, the data filled by user in any Activity or fragment may be lost if the focused screen is interrupted. Data should be saved and restored using overridable methods onSaveInstanceState and onRestoreInstanceState.

Context: UI
Affects: User Experience
Problem: The missing implementation of onSaveInstanceState(Bundle) and onRestoreInstanceState(Bundle).
Refactoring: Implement and proper use of both onSaveInstanceState(Bundle) and onRestoreInstanceState(Bundle) in Activity or Fragment.
Resolves: User Experience and User data security.
Detection Rule:
(Search Activity or Fragment in project source code) AND (Check missing implementation of any two callback methods onSaveInstanceState and onRestoreInstanceState).

15 Untouchable

In Android applications, any touchable UI view should be greater than 48dp because the view with less than 48dp is not easily touchable.

Context: UI
Affects: User Experience
Problem: It is hard for a mobile user to tab on Button whose width, height i.e. size is less than 48dp.
Refactoring: Increase the size of all touchable views up to 48dp.
Resolves: User Experience and accessibility.
Detection Rule:
(Search the touchable views i.e. Button, ImageButton, and Check Box etc.) AND(Check the size of touchable view is less than 48dp w.r.t. width or height)

.

16 Uncontrolled Focus Order

The usage of default (up, down, right and left directional) control for navigation is not recommended as in some cases it might be illogical.

Context: UI
Affects: User Experience
Problem: User may irritate while using the mobile application as the navigation control is not according to the his/her desire or illogical.
Refactoring: Use all four directional controls as nextFocusDown, nextFocusUp, nextFocusRight and nextFocusLeft.
Resolves: User Experience and accessibility.
Detection Rule:
(Search UI View type element in layout XML file) AND (Finding the UI views with missing navigation controls).

17 Nested Layout

In Android UI, the layouts having elements with attribute weight must be computed twice. This particular phenomenon increases the computation time exponentially. Mostly, the LinearLayouts creates that problem.

Context: UI, efficiency
Affects: User Experience

Problem: Deep nesting in layouts consumes computation power of mobile device and effects user experience due to late rendering.
Refactoring: Use Flattened UI instead of deep nested layouts. This can be achieved by replacing LinearLayout with RelativeLayout or CoordinateLayout.
Resolves: User Experience and accessibility.

Detection Rule:

(Searcch LinearLayouts in xml file.) AND(Search LinearLayouts with weight attribute.)

18 Not Descriptive UI

Android provides an app named as TalkBack used for visually impaired people to keep track of navigation and keep track of applications that user is currently using. TalkBack maintains navigation track by reading the contents which are defined as contentDescription in UI views. Missing of contentDescription indicates poorly descriptive UI.

Context: UI
Affects: User Experience
Problem: The Android provided app TalkBack fails to track the user navigation and the application may useless for visually impaired people.
Refactoring: Use contentDescription attribute in every UI element
Resolves: User Experience and accessibility.

Detection Rule:

19 Set Config Changes

Android provides default implementation for orientation changes i.e. from portrait to landscape and vice versa. The usage of default implementation for orientation changes using the attribute android:configChanges may cause memory leaks leaving resources in memory.

Context: Memory efficiency, UX

Affects: User Experience, Memory

Problem: It may memory intensive task to use default handling of screen orientation.

Refactoring: Use onConfigurationChanged(Configuration newConfig) instead of android:configChanges.

Resolves: User Experience and accessibility.

Detection Rule:

(Search activity elements in AndroidManifest.xml file) AND (Search activity elements with attribute android:configchanges).

20 Overdrawn Pixel

In Android application design, most of the UI part is built using XML layouts that consist of a stack of several UI elements. In a stack of UI elements, it might be possible to overdraw a pixel multiple times using Android:background property which is a GPU extensive task.

Context: UI, Efficiency
Affects: User Experience, Efficiency
Problem: Stacking views leads to overdrawn pixels' multiple times which is a resource intensive phenomenon.
Refactoring: Just like Nested Layout Overdrawn pixel can be refactored with flattened UI.
Resolves: User Experience, efficiency and startup time.

Detection Rule:

(Search Layout with attribute android:background and having children more than one) AND (Layout children with attribute android:background).

21 Static Context

In Android applications, declaring field of type Context leads to memory leak as it is never collected by garbage collector of JVM.

Context: Memory, Efficiency
Affects: User Experience, Efficiency
Problem: Using static field of type Context consumes a lot of memory as it holds the reference of Activity or application.
Refactoring: Use interface in order to get context or pass a parameter of type Context.
Resolves: User Experience, Memory efficiency.
Detection Rule:
(Finding the static field of type Context in Java class.)

22 Static Views

Holding static reference of any view type like Button, ImageView, View, EditText, TextView or any Layout is a resource (memory) intensive process.
Context: Memory, Efficiency
Affects: User Experience, Efficiency
Problem: Using static field of any View type is a resource (memory) intensive task.
Refactoring: Access the view by implementing the interface or passing view as a parameter.
Resolves: User Experience, Memory efficiency.
Detection Rule:
(Search static field of type View like LinearLayout, RelativeLayout, Button, View, ImageView or TextView etc in Java class.)

23 Static Bitmap

Holding static reference of type Bitmap is a resource (memory) intensive process as bitmaps are heavy objects and they consume a lot of memory.
Context: Memory, Efficiency
Affects: User Experience, Efficiency
Problem: Using static field of Bitmap is a resource (memory) intensive task.
Refactoring: Access the bitmap by implementing the interface or passing it as a parameter.
Resolves: User Experience, Memory efficiency.
Detection Rule:
(Search static field of type Bitmap in Java class)

24 Collection of Views

The collection of views is a resource intensive process in Android applications and should be avoided.
Context: Memory, Efficiency
Affects: User
Experience, Efficiency
Problem: Using collection of views is a resource (memory) intensive task.
Refactoring: Use layout inflator with xml layouts instead of views collection.
Resolves: User Experience, Memory efficiency.
Detection Rule:
(Search Collection type like List, ArrayList, LinkedList with type parameter View type like Button, ImageView and Textview etc)

25 Collection of Bitmaps

The collection of Bitmaps is a resource intensive process and should be avoided as it also leads to memory leak.
Context: Memory, Efficiency
Affects: User Experience, Efficiency
Problem: Using collection of bitmaps is a resource (memory) intensive task.
Refactoring: Use drawables and store their references (Integer Constants) instead of Bitmaps.
Resolves: User Experience, Memory efficiency.
Detection Rule:
(Search Collection type like List, ArrayList, LinkedList with type parameter type Bitmap)