10-2016

Are Java Programming Best Practices Also Best Practices for Android?

Yoonsik Cheon
University of Texas at El Paso, ycheon@utep.edu

Comments:
Technical Report: UTEP-CS-16-76

Are Java Programming Best Practices Also Best Practices for Android?

Yoonsik Cheon

TR #16-76
October 2016

Keywords: Android, garbage collection, memory allocation, Java, object-oriented programming, programming practices.

1998 CR Categories: D.1.5 [Programming Techniques] Object-oriented Programming; D.2.2 [*Software Engineering*] Design Tools and Techniques—object-oriented design methods; D.2.3 [*Software Engineering*] Coding Tools and Techniques—object-oriented programming; D.3.3 [*Software Engineering*] Language Constructs and Features—classes and objects, dynamic storage management; D.4 [*Operating Systems*] Storage Management—garbage collection

Department of Computer Science
The University of Texas at El Paso
500 West University Avenue
El Paso, Texas 79968-0518, U.S.A

Are Java Programming Best Practices Also Best Practices for Android?

Yoonsik Cheon
Department of Computer Science
The University of Texas at El Paso
El Paso, Texas, U.S.A.
ycheon@utep.edu

Abstract—Android apps are written in Java. Android beginners assume that Java programming best practices are equally applicable to Android programming. In this paper, we perform a small case study to show that the assumption can be wrong. We port a well-written Java application to Android. A certain key assumption of object-oriented programming doesn't hold on the Android platform. Thus, some of the best practices in writing Java programs are not best practices for Android. In fact, they are anti-patterns that Android programmers should avoid. We show concrete examples of these anti-patterns or watch-outs along with their fixes.

Keywords—garbage collection, memory allocation, object-oriented programming, programming practices, Android, Java.

I. Introduction

Android is one of the most popular mobile platforms paving the way for the development of a flood of mobile apps. Android apps are written in Java though there is some difference between the Java API and the Android API. So, many programmers think that Android development is easy for Java programmers. They also think it's important to understand and learn Android because of its dominance in mobile computing. In fact, a significant number of Java programmers start Android programming right away after reading a few tutorials. They expect to carry over the best coding practices (e.g., [2] [9]) that they learnt and mastered through their Java programming.

Android devices are resource-constrained with storage capacity and battery lifetime, and performance is always a problem for anyone developing Android apps [7]. Memory, for example, is a lot more valuable on Android than on other operating systems. An application launched on Microsoft Windows, for example, may stay running indefinitely. It's different on Android in that it has a memory conservation mechanism known as low memory killer (LMK). When too much memory is used, LMK will start killing background and inactive processes consuming large amounts of memory. In short, Android apps are expected to use memory more efficiently. And thus Android programmers need to build them with memory conservation in mind.

One key assumption of object-oriented programming is that objects are cheap. They don't take lots of resources such as memory. Their applications (sending messages to objects) are as efficient as procedure calls. The idea is to let many small objects solve a task together, each object focusing on a small aspect of the task. Unfortunately, this key assumption of object-oriented programming doesn't hold on the Android platform. Android devices have limited storage capacity.

The author has been teaching a mobile application development course for several years. The entering students are already familiar with the object-oriented design and programming using Java. They have at least two years of programming experience in Java, including CS 1, 2, and 3. At first, most students think that writing good Android code is the same as writing good Java code. Perhaps, most beginning Android programmers have a similar mindset.

In this paper, we perform a small case study to show that it is indeed a misconception. The students taking the above-mentioned mobile apps course have already learned many good Java programming tips, guidelines, and styles that they can incorporate into their daily programming as general coding practices. These best practices can transform a piece of Java code into an excellent program. We show that not all good Java coding practices are applicable to the development of well-behaved Android apps. We focus our discussions on memory efficiency for two reasons. Because of limited physical storage, it is one distinguishing feature of mobile platforms. It is also a topic often neglected early in a Computer Science curriculum. This is especially true in a Java-based one due to the automatic memory management of Java.

As our case study we use Battleship game (see Section II). It is the kind of programs that our students are expected to write in our object-oriented programming and design courses as well as the mobile app development. We first describe a Java application, coded by following the recommended coding practices of Java (e.g., [2] [9]). We then port it to Android by reusing as much code as possible. We redesign only the UI part by using the Android framework classes; the functional core code remains the same. We measure the memory performance of the Android app. We then study its code to learn about the impact of the Java coding practices inherited from the Java application. We finally refactor the code to improve its memory efficiency.

It is not our goal to provide an extensive or comprehensive list of guidelines for creating high performance Android apps

[1] [8] [10]. We don't either intend to propose a new technique for identifying and removing so-called *code smells* [3] [9] [11]—bad implementation practices within Android apps that may lead to poor software quality, in particular performance [5]. We would like to caution Android beginners that they have to keep an eye on and be proactive in reducing memory usage within their apps. They often need to be cautious in applying even the most obvious and intuitive coding practice of Java (see Section IV).

In the next section we describe the target of our exercise, a Battleship game application written in Java.

II. Battleship Game

In this paper we will consider a Java application that allows a user to play Battleship games (see Figure 1). Battleship is a well-known guessing game for two players. The game is played on grids, usually 10×10, of squares. Each player has a fleet of ships, and each ship occupies a number of consecutive squares on the grid, arranged either horizontally or vertically. Once the ships are secretly positioned on the grids of the players, the game proceeds in a series of rounds. In each round, each player takes a turn to make a shot to a square in the opponent's grid. A shot is either a 'hit' on a ship or a 'miss'. When all the squares of a ship have been hit, the ship sinks. If all of a player's ships have been sunk, the game is over and the opponent wins.

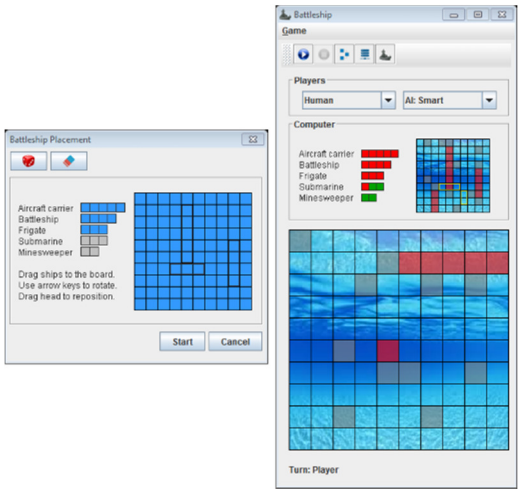

Figure 1. Screenshots of Java application

Figure 1 shows sample screenshots of the Battleship application written in Java. The main GUI is shown on the right and the one on the left is a dialog for placing a player's fleet of ships on the opponent's board. The application provides a few different playing modes. But, we will consider only the strategy mode in which a user plays a game against the computer by selecting one of the computer move strategies. The application consists of 19 classes and 2782 lines of source code, counting only those that are concerned with the strategy mode.

Figure 2 shows the design of the Battleship application. The class diagram describes only the model classes and their relationships, which can be reused on the Android platform. The GUI classes such as dialogs and special panel classes to display game boards using 2D graphics are not included. They will be redesigned and recoded using the Android framework classes (see Section III.A).

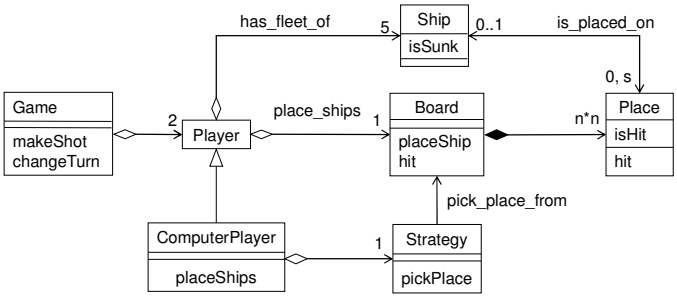

Figure 2. Class diagram

In the next section we will show an Android app version of the Battleship game that we created by reusing the code of the Java application.

III. Android App

A. *Our Approach*

We implemented an Android version of the Battleship application by reusing as much Java application code as possible. Figure 3 shows two screenshots of the app: placing ships (left) and making shots (right).

Figure 3. Screenshots of Android app

As expected we were able to reuse all the model classes with only minor changes such as getting rid of Java 8 features. Android doesn't yet support such Java 8 features as lambda expressions and streams. Most of our efforts thus were on designing and coding the GUI using the Android framework classes. We defined several Android activity and view classes. An *activity* is an Android app component for a single screen. It is in a sense a unit of Android programs in that an app usually consists of one or more activities. If an app consists of two screens, the standard approach is to create two activities, each

with its own life cycle. A *view* is the basic unit of the Android UI and represents a widget that has an appearance on the screen. An activity is composed of views. Figure 4 shows UI classes along with model classes that are referenced, including:

- Three activity classes: *MainActivity*, *FleetActivity*, and *PlayActivity* denoted with the *activity* stereotype. The first class is for selecting a play mode such as a strategy game. The last two classes are for deploying ships and making shots.
- A hierarchy of view classes to display game boards using 2D graphics. As depicted, there are three concrete board view classes: (a) the *FleetBoardView* class for deploying the player's ships, (b) the *BoardView* class for displaying the player's board with revealed ships to show shots made by the opponent, and (c) the *TouchBoardView* class for displaying the opponent's board to let the player make shots and show them. The last class, of course, doesn't reveal the hidden ships unless they are hit or sunk, and it responds to screen touch events.

Even though all these classes are coded with Android-specific framework classes, their display logic and algorithms are identical to those of the Java application. For example, all board view classes display their boards in three steps: (a) paint the background, (b) display a 2D grid by drawing horizontal and vertical lines, and (c) iterate over all the places of the board and check the presence of ships on them to draw special markers on those places that were hit. Because of the last step, the performance of a board view class may be influenced by the way the board class is coded (see Section IV).

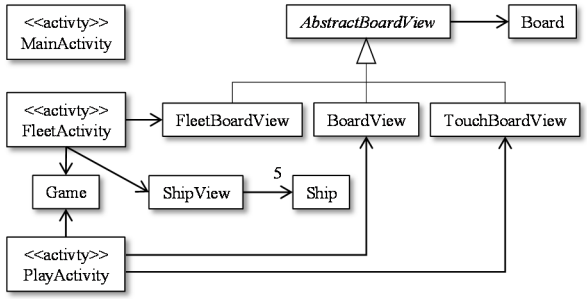

Figure 4. UI classes

One minor change to the model classes is to use the Singleton design pattern [4] for the *Game* class to share its instances among activities. Recall that each activity has its own lifecycle even if they may run in the same process. We need to ensure that the *PlayActivity* class uses the same game object as the *FleetActivity* class creates and uses. The Singleton design pattern provides a simple way to share data among activities of the same app.

B. Memory Usage

To learn about the memory usage of our app we used the profiling tools included in the Android Studio. We played one strategy game and monitored the memory usage of our app in real-time. We observed the changes in the amount of memory allocated to our app at the main steps of playing a game. Playing a game requires the following three steps:

1. Select a play mode (*MainActivity*)
2. Deploy ships (*FleetActivity*)
3. Make shots (*PlayActivity*).

Figure 5 shows the memory usage of our app. The top graph depicts changes in the allocated memory when the above three activities are launched or started to play a game. When the fleet activity is started, the allocated memory increases sharply from 5.45 MB to 12.98 MB. After that the memory use increases steadily in a linear fashion even if there is no user interaction. Similarly, placing all the ships at once randomly and starting the play activity cause memory use to rise sharply. We also learned that each shot costs approximately 0.02 MB of dynamic memory, and sinking a ship requires a few megabytes of dynamic memory. The bottom graph depicts the garbage collection cycle of the play activity. As noted before, the memory use increases in a linear fashion even if the app is idle without interacting with the user. This causes a garbage collection at every 20 seconds. The fleet activity's garbage collection cycle is approximately 50 seconds. Most of the steadily allocated memory are in fact garbage. After a (manually-forced) garbage collection, the amount of allocated memory drops near to the initial level, e.g., 13.10 MB for the fleet activity.

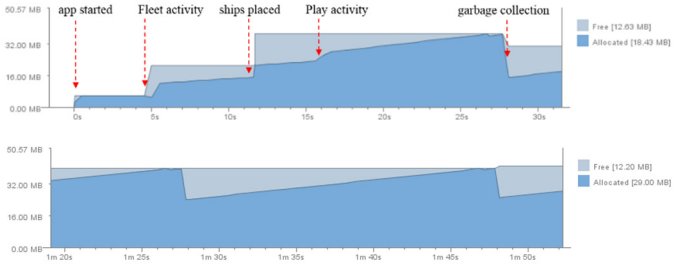

Figure 5. Memory usage of the app (top) and the PlayActivity class (bottom)

One might think that for an app like Battleship, collecting garbage at every 20 or 50 seconds may not be an issue. However, minimizing the garbage collection execution time is important because garbage collection generally results in poor performance of the app and the overall slowdown of the system; the app may be suspended during garbage collection [7]. It is also an issue because of such side-effects as battery consumption and heat. Recall that battery lifetime is an invaluable resource on Android devices. A study has shown that Android's garbage collection consumes a significant amount of energy [6]. A frequent garbage collection therefore may drain the battery faster. It also produces more heat, causing a device to become warm and hot, sometimes to the point it becomes unusable. In fact, we observed that our smartphones became somewhat hot in less than 10 minutes of use after launching the

app. A well-behaved Android app should use as little battery as possible.

In the next section we will study why our app gives such a poor memory performance. We describe several types of memory related *code smells* present in our code and refactor them. A memory code smell is a bad implementation practice that may lead to poor memory performance [3] [11].

IV. Refactoring App Code

In this section we refactor the Android app code to improve its memory efficiency. We show several code snippets picked from our app. They are all well-written by following the conventional wisdom and coding practices of Java. We scrutinize them for a *code smell*, a bad coding practice that may lead to poor memory performance. Our sample code includes the following:

- Activities
- Iterators
- Enhanced *for* statements
- Choice of data structures
- Local variables
- *onDraw*() methods
- Releasing objects

Activities: An Android app consists of several app components such as activities and services. Each component has its own lifecycle even if it may run in the same process as other components. It may also have a different memory need. This is different from a Java application where parts share a global memory space. Our app consists of three different activities: *MainActivity*, *FleetActivity*, and *PlayActivity* (see Section III.A). We can consider the memory use of each activity, the reason being that only one activity runs at a time. The one at the top of the so-called *activity stack* is active. In a sense we can partition the memory need of the app among activities. We did such an optimization for our app by refactoring the *AudioEffect* class. The class loads and plays all five different audio clips for the whole application. As said before, there is one global memory used by each part of the application. In the Android app, however, no activity need all five audio clips. For example, the *FleetActivity* class uses only one audio clip. Thus, one way to conserve memory is to load audio clips for each activity. The table below shows the memory usage of the app before and after our refactoring. Note that the amount of memory allocated for the *FleetActivity* class drops from 13.09 MB to 9.92 MB, 24% improvements.

	Allocated memory (MB)		
	Main	Fleet	Inc (F – M)
Before	4.18	17.27	13.09
After	4.92	14.84	9.92

Iterators: Java provides an interface named *Iterator*. It implements the Iterator design pattern [4] to iterate over elements of a container or aggregate object. The interface defines methods like *hasNext*() and *next*(). The use of an iterator for an aggregate is a recommended, good coding practice. It provides a way to access the elements of an aggregate without exposing the underlying representation. The implementation details of the aggregate such as data structures and algorithms are not exposed to the client code. The Battleship application has several 1-to-many associations (see Figure 2). As expected, its code uses iterators to make the elements of the aggregates accessible. For example, the *Board* class define a method named *places* to provide sequential access to all its places. Note that the method returns an *Iterable* object. It is a new interface introduced in Java 8 to support the for-each statement (see below), but the idea is the same.

```
private final List<Place> places;
public Iterable<Place> places() {
  return places;
}
```

One potential problem with the use of the *Iterator* interface is that each call may result in the creation of a new iterator object. If a critical section of code calls the method, it may end up creating many temporary objects. An example of critical code is the *onDraw*() method of a view class (see below). The Android system calls the method to refresh the screen up to 60 times per seconds [10]. This is because the screen refresh rate of most Android device is 60 Hz. Indeed, this is the case for the Battleship application. For example, the *BoardView* class handles displaying a board. Its *onDraw*() method calls the above *places*() method to display the current state of the board. Our fix is to drop the use of iterator objects. Instead we provide direct accesses to the underlying representations such as lists and arrays.

Enhanced *for* statements: Related with the iterator object is the use of the enhanced *for* statements. The statement is also called "for-each" statement and is introduced in Java 5. It is used to iterate all elements of an array or an *Iterable* object, including a collection. Since it provides a simpler way to iterate through elements, its use is recommended. In fact, it is a popular feature. The code snippet below shows an example use of it in the Battleship application. The method is from the *FleetBoardView* class providing a UI for deploying a fleet of ships of the player.

```
protected void onDraw(Canvas canvas) {
  super.onDraw(canvas);
  for (Ship ship: fleet) {
    if (ship.isDeployed()) { /* … */ }
    // …
  }
}
```

The use of for-each statements produces concise and more readable code. In the above code, for example, one doesn't have to manipulate an index or loop variable to access all the ships contained in *fleet*. One doesn't have to know whether *fleet* is an array, a collection, or an *Iterable* object. But, the above, fine looking code have a potential problem. Each execution of the for-each statement may create a new iterable/iterator object. And it could be an issue if the statement occurs inside a critical

section of code such as the *onDraw*() method of a view class. Recall that the *onDraw*() method may be called up to 60 times per second. With an *ArrayList* the for-each statement is also said to be about 3 times slower than the for statement [1]. Our fix is to get rid of the use of for-each statements. We replace them with the traditional *for* statements, as shown below.

```
for (int i = 0; i < fleet.length; i++) {
  final Ship ship = fleet[i];
  if (ship.isDeployed()) { /* … */ }
  // …
}
```

Choice of data structures: In any non-trivial application, there will be associations with multiplicity more than one. In the Battleship application, for example, a game consists of two players, a player has a fleet of five ships, and a ship is placed in a sequence of places. A board consists of $n \times n$ places. One has several choices in representing these 1-to-many associations. Two most common choices are arrays and collections like *HashSet*, *HashMap*, *ArrayList*, and *LinkedList*. A general guideline is to use appropriate collections rather than arrays [2]. A collection class like *ArrayList* provides a well-defined interface to access and manipulate elements. The client can't see the hidden internal representation. An array, however, defines a structure to be manipulated directly by the client. This difference is highlighted by the two different versions of the *removePlace*() method of the *Ship* class, shown below.

```
private List<Place> places;
public void removePlace(Place place) {
    places.remove(place);
}

private Place[] places;
public void removePlace(Place place) {
   for (int i = 0; i < places.length; i++) {
      if (places[i] == place) {
         for (int j = i; j < places.length - 1; j++) {
            places[j] = places[j + 1];
         }
         places[places.length - 1] = null;
         break;
      }
   }
}
```

The collection classes generally provide better support for manipulating elements at a higher abstraction level. Its use produces code that is concise and easy to understand. Inside performance critical code, however, one may need to reconsider the above guideline of preferring collections over arrays. In general, working with arrays is the fastest possible. For example, iterating over an array is significantly faster than iterating over an *ArrayList*. Arrays can also be more memory efficient. And unlike collection their memory use is predictable. For collections, one need to watch for dynamic memory allocations. There may be hidden internal objects such as node objects of linked lists. Another common case overlooked by many beginning Java programmers is collections of primitive values. For example, every time an *int* value is added to an *ArrayList<Integer>* collection, a new *Integer* is created due to auto-boxing. The Java compiler makes an automatic conversion between primitive types and their wrapper classes. Thus, an *int* array is definitely the winner in a memory-concerned critical code.

It is also possible to make the use of a collection more memory efficient. The simplest and easiest way to conserve memory is to use a bounded collection. One can specify the initial capacity of an array-based collection such as *ArrayList*. It will improve the execution time as well. However, it assumes that one knows the fixed size of a collection in advance. This is the case for the *Board* class, and its constructor may initialize places and ships as follows.

```
places = new ArrayList<>(size * size);
ships = new ArrayList<>(5);
```

Local variables: A good coding practice is to reduce the scope of local variables [2]. The idea is to reduce the scope of a variable so that it is only visible in the scope where it is used. Such a variable won't clutter the name space, prevents from introducing an unused variable, and makes the code more readable. The Battleship application code adhered to the coding practice, as shown in the following code snippet of the *ShipView* class.

```
protected void onDraw(Canvas canvas) {
  super.onDraw(canvas);
  Paint shipPaint = new Paint();
  if (ship.isDeployed()) { /* … */ }
    Paint xPaint = new Paint();
    float[][] xLine = new float[][] { … };
    // … canvas.drawLines(…, xPaint);
  }
  // …
}
```

A ship is drawn using a paint stored in a local variable named *shipPaint*. If a ship is deployed, an X mark is shown at its head section (see Figure 1). For this, the *if* statement introduces two more variables *xPaint* and *xLine* whose scopes are the body of the *if* statement. All three local variables are declared in the scope that they are used, by following the above coding practice about local variables. However, one serious concern with the above code snippet is that three new objects are created each time it executes. Recall that an *onDraw*() method may run up to 60 times per second. A simple and obvious tip to conserve memory is to avoid creating unnecessary, temporary objects [2]. In a critical section of code, one should avoid using local variables to store dynamically created objects. Instead, if possible, one should use fields to cache and reuse the dynamically created objects, as shown below. The refactored code clutters the name space but is more memory efficient.

```
private final Paint shipPaint = new Paint();
private final Paint xPaint = new Paint();
private float[][] xLine = { new float[2], new float[2] };
```

onDraw() methods: It is obvious that the most benefit is obtained by optimizing code that runs frequently. One such a method is the *onDraw*() method of a view class, which handles drawing a view on the screen. This method is called by the Android system when it needs to refresh the screen. On most Android devices, rendering is usually done at 60 fps; the devices refresh the screen 60 times per second [10]. Therefore, it is crucial to ensure that all the rendering can occur in less than 16 ms (1 s / 60 fps = 16 ms). Each frame has 16 ms to be handled, including its drawing. In general, the method should avoid any operation or logic other than drawing. Furthermore, allocating objects in this method can have a devastating effect on garbage collection. For example, we can avoid creating new *Paint* objects by creating them once in the constructor. The idea is to store them in the fields and reuse on every invocation of the *onDraw*() method. This is exactly what we did above.

Releasing objects: As stated earlier, the best way to conserve memory is to avoid creating unnecessary objects. And, the second best way is to release objects when they are no longer needed so that they can be collected by the Android garbage collector. The guiding practice is to eliminate obsolete object references [2]. In general, an object is eligible for garbage collection when there is no reference to it. And thus, it's a good practice for a large object to provide an explicit way to release its resources and make it eligible for garbage collection. As mentioned earlier in this section, we refactored the *SoundEffect* class. One refactoring is to introduce a method to release all audio clips loaded for an activity. An activity calls the method when the loaded audio clips are not needed. An activity doesn't need the audio effect when, for example, it is paused and stopped. The following code shows an example use of the *release*() method by the *FleetActivity* class. Upon starting the *PlayActivity* class, the code release all loaded audio clips.

```
public void doneClicked(View view) {
   if (allShipsPlaced()) {
      startActivity(new Intent(this, PlayActivity.class));
      soundEffect.release();
   } else { /* … */ }
}
```

V. Evaluation

The main problem with the initial version of our app is frequent garbage collections. As depicted in Figure 5 of Section III.B, the memory use of the app increases in a linear fashion. It leads to frequent garbage collections, once at every 20 or 50 seconds. Recall that this happens even if the app is inactive, without interacting with the user. It also causes overheating of the device.

Is the problem fixed with the code refactoring done in the previous section? Figure 6 shows the memory usage of the refactored code (compare it with Figure 5 in Section III.B). Once an activity starts, the graph becomes almost flat, meaning no dynamic memory allocation. Of course, there is still some dynamic memory allocations, e.g., for making shots and sinking ships. But, completing a game doesn't require a garbage collection. It would be fair to mention that the amount of initial memory allocation increases about 36%. We believe that this is due to caching of objects, which are before created on-the-fly and stored in local variables. The app can now run continuously without causing a heating problem. What is the reason that our refactoring becomes effective? We focused on the code that is called by the *onDraw*() methods. It is so-called *critical code* in that the Android system may call it at every 16 milliseconds on a 60 Hz device; it redraws the activity.

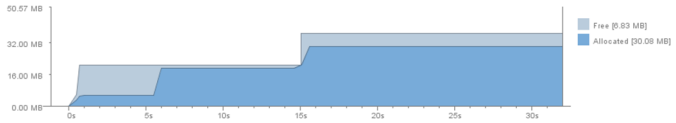

Figure 6. Memory usage of the refactored code

VI. Conclusion

Android devices are resource-constrained with storage capacity and battery lifetime. Thus, performance is always a problem for anyone developing Android apps. One performance concern is memory efficiency, which is often neglected in Java due to garbage collection. We showed that some of the well-known Java programming practices can be a source of a memory problem. For example, even the enhanced *for* statement may cause a significant memory overhead. Thus, one should not only be cautious in applying even well-known Java coding practices but also scrutinize any critical section of code, code such as the *onDraw*() method of a view class. One should use profiling tools to gain insight into, find, and fix a problem within one's app.

References

[1] *Best Practice for Performance*, https://developer.android.com/training/best-performance.html

[2] J. Bloch, *Effective Java*, second edition, Addison-Wesley, 2008.

[3] M. Fowler and K. Beck, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999.

[4] E. Gamma, et al., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.

[5] G. Hecht, N. Moha, and R. Rouvoy, An empirical study of the performance impacts of Android code smells, *International Conference on Mobile Software Engineering and Systems*, Austin, TX, 2016, pages 59-69.

[6] A. Hussein, et al., Impact of GC design on power and performance for Android, *Proceedings of the 8th ACM International Systems and Storage Conference,* Haifa, Israel, pages 13:1-13:12, 2015.

[7] M. Linares-Vasquez, et al., How developers detect and fix performance bottlenecks in Android apps, *IEEE International Conference on Software Maintenance and Evolution*, September 2015, pp 352-361.

[8] E. L. Manas and D. Grancini, *Android High Performance Programming*, Packt Publishing, 2016.

[9] R. C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*, Prentice Hall, 2008.

[10] D. Sillars. *High-Performance Android Apps: Improve Ratings with Speed, Optimizations, and Testing*, O'Reilly, 2015.

[11] M. Tufano, et al., When and why your code starts to smell bad, *IEEE International Conference on Software Engineering,* 2015, pages 403-414.