10-2020

Coding Overhead of Mobile Apps

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

Comments:

Technical Report: UTEP-CS-20-106

Coding Overhead of Mobile Apps

Yoonsik Cheon

TR #20-106
October 2020

Keywords: code complexity; mobile app, Android, Java

2012 ACM CCS: • Software and its engineering ~ Software creation and its management • Software and its engineering ~ Software notations and tools • Human-centered computing ~ Mobile computing

Submitted for publication.

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

Coding Overhead of Mobile Apps

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

***Abstract* — A mobile app runs on small devices such as smartphones and tablets. Perhaps, because of this, there is a common misconception that writing a mobile app is simpler than a desktop application. In this paper, we show that this is indeed a misconception, and it's the other way around. We perform a small experiment to measure the source code sizes of a desktop application and an equivalent mobile app written in the same language. We found that the mobile version is 19% bigger than the desktop version in terms of the source lines of code, and the mobile code is a lot more involved and complicated with code tangling and scattering. This coding overhead of the mobile version is mostly due to the additional requirements and constraints specific to mobile platforms, such as diversity and mobility.**

Keywords—code complexity, mobile app, Android, Java

I. Introduction

Mobile platforms are one of the most popular application platforms of today alongside the Web platform. It is said that people are spending more time on mobile devices than on desktop or laptop computers, and the number of mobile app downloads has been steadily increasing. There are nearly six million mobile apps available today through app stores [5]. Mobile app development is growing fast, and now represents a significant part of the software industry. And thus, many are thinking about learning how to code mobile apps, and some take the plunge.

There is, however, a common misconception that coding a mobile app is simpler than a desktop application. Perhaps, this is originated from the fact that a mobile app runs on small devices such as smartphones and tablets. It may also be caused by the fact that mobile apps are smaller than traditional applications with the average size of 5.6K source lines of code (SLOC) [4], and the development of mobile apps tend to be driven by a single developer or a small team [6].

In this paper, we show that it is indeed a misconception by studying the additional code besides the core business logic that has to be written for mobile apps. For this, we perform a small experiment in which we write two versions of an application in the same language, one for a desktop platform and the other for a mobile platform. We then measure the source code sizes of the two versions as well as studying code that appears only in the mobile version. Our finding is that the mobile version is 19% bigger than the desktop version in SLOC. In addition to performing the core business logic of the application like the desktop version – in fact, the business logic code is reused, or shared, between the two versions – the mobile app code has to meet the mobile platform-specific requirements and constraints. For example, 17% of the mobile app code was written to handle screen orientation changes. When a mobile device is rotated and the screen changes orientation, say from portrait to landscape, the app needs to present the same contents as before but in the landscape layout.

The main contribution of our work is a quantitative measurement of the amount of code that has to be written to address mobile platform-specific requirements and constraints. Mobile apps have different characteristics from traditional desktop applications [4], and developing mobile apps presents different challenges compared with desktop apps [3]. Platform differences are the primary source of challenges in developing applications for multiple platforms. It was shown, for example, the application programming interface (API) differences are the major factor that determines the amount of code reuse possible between Java and Android Java [1]. We, however, found no published work measuring the coding overhead of mobile apps compared with equivalent desktop applications.

The rest of this paper is structured as follows. In Section II we describe our experiment approach including the application to be created – an app to track fluctuating prices of products, scraped from Web pages. In the next two sections, we design and code a Java desktop version and an Android native app written in Java, respectively. In Section V, we compare the two versions by measuring their code sizes and analyzing the additional code written for the Android version. In Section VI, we conclude our paper with a concluding remark.

II. Our Approach

Our approach to studying the coding overhead of mobile applications is a small experiment in which we create two versions of an application, one for a desktop platform and the other for a mobile platform. We then analyze and compare the source code of the two versions. In this paper, we use the term *platform* in a very narrow sense to mean a software development kit (SDK) including its application programming interfaces (API). Besides the platform, other factors may affect the source code size of an application, and thus we need to control and minimize the influence by factors other than the platform. The influence may be internal in the sense it is due to the application itself or external to the application.

An application itself can of course influence its source code complexity. For example, even if two versions of an application provide the same functionality, different user interfaces may result in different code complexities. The programming languages also affect the source code size of an application, as some programming languages are more expressive than others in the sense that they provide more succinct notations and are

less verbose. In our experiment, therefore, we code both versions of the application in the same programming language. We also implement similar user interfaces for the two versions modulo the graphical user interface frameworks of the underlying platforms. However, to have a realistic experiment, the user interfaces will be designed by following the guidelines, styles, and conventions of the platforms even if there are differences between the two versions, e.g., the use of the swipe gesture in the mobile version.

We develop an application named Price Watcher that tracks the prices of products, or items, extracted from their Web pages. Actually, it was written for another study [1] and will be adapted a bit in this paper. The application helps a user to figure out the best time to buy items by watching over fluctuating prices. Since the prices are scraped from Web pages, the watch list may consist of items from different online stores or websites. We create a Java desktop application as well as a native Android mobile app written in Java. By coding both versions in Java, we eliminate the language difference concern mentioned above and can reuse significant code between the two versions despite the platform differences.

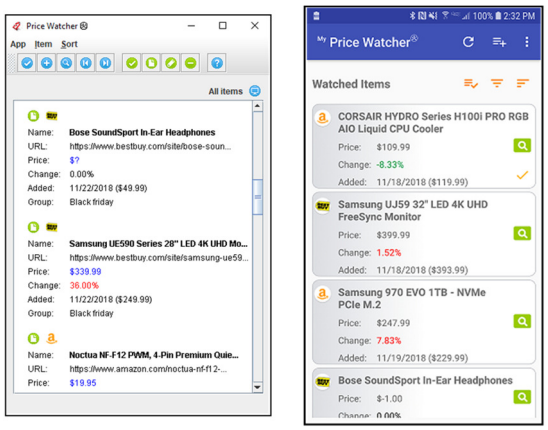

Figure 1. Screenshots of Price Watcher (Java and Android)

Figure 1 shows the screenshots of both versions of the application. By having similar user interfaces, we minimize the influence of the user interface designs on the source code complexity. We display multiple items in a scrollable list, and each item is manipulated with a popup menu of which a couple of frequently used menu items are shown as image icons for quick access. The drop-down menus at the top of the screen provide display-related operations applicable to all items, such as searching, filtering, grouping, and sorting the items. In the Java version, the main menu bar and toolbar provide operations such as adding new items and checking the prices of existing items. In the Android version, these operations are provided as the app bar, one of the important user interface design elements of an Android app.

III. Desktop Application

The desktop version is a Java application written using the Java AWT/Swing graphical user interface frameworks. To make the application realistic, we use an SQLite database to store the items being watched. SQLite is a serverless relational database system, meaning that it is embedded into the application.

Figure 2 shows the design of the application – main classes and their relationships. We use the model-view-controller (MVC) design to separate the business logic from the user interface. This design not only modularizes our application but also allows us to reuse the business logic classes in the Android version (see Section IV). The top half of the diagram shows the user interface classes. These are custom widget classes such as ItemView, ItemFilter, and dialogs as well as view-specific model classes such as ItemListModel. We display a collection of items with a list widget (JList), but each item is displayed by the ItemView class. The subclass ItemCellRender adapts the ItemView class to work with the list widget. The ItemListModel class is an adaptor to provide the items being watched to the list widget.

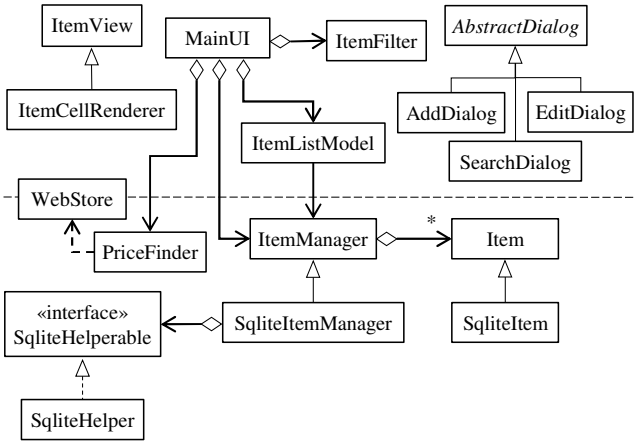

Figure 2. Design of the Java version

The bottom half of the diagram shows the business logic, or model, classes including data, persistence, and network classes. An item being watched is represented by the Item class, and the ItemManager class manages a collection of items. A special subclass named SqliteItemManager persists items by storing them in an SQLite database. For this, the SqliteItem extends an item state to store SQLite-specific information such as the primary key. An interesting design decision is the introduction of an interface named SqliteHelperable. The SqliteItemManager class accesses – stores, reads, and updates – items in the database through this interface. There are some API differences between Java and Android implementations of SQLite, and the interface hides these differences to the manager class so that it can be reused in the Android version of the application as well (see Section 0). The SqliteHelper class implements the interface by using the Java implementation of SQLite. For example, it connects to the database with a Java Database Connectivity (JDBC) driver. The PriceFinder and WebStore classes are responsible for networking and Web scraping. Actual work is done by the WebStrore class, which is an enum type and defines all the websites supported by the application. For each website, it provides information such as store name, URL, and icon as well as an algorithm to parse and extract an item's price from a

Web document. Since each website displays the price of an item differently, different Web scraping algorithms are defined for different websites.

The implementation of the application consists of 22 Java classes with 3157 source lines of code (SLOC) including program comments (see Table 1). About 68% of the source code is concerned with the user interface of the application. This is not surprising, as the business logic of the application is somewhat straightforward, and the user interface is coded programmatically in Java. The business logic consists of three parts: (a) managing the list of items, (b) storing them in a database, and (c) finding the current prices of the items by scraping their Web pages. Each of these business logic parts contributes about 10~11% of the total lines of source code.

Table 1. Code size of the Java version

Classes	**No. of Classes**	*Percent (%)*	**No. of Lines**	*Percent (%)*
UI	14	63.64	2145	67.94
Data	2	9.09	348	11.02
Storage	4	18.18	345	10.93
Network	2	9.09	319	10.10
Total	22	100	3157	99.99

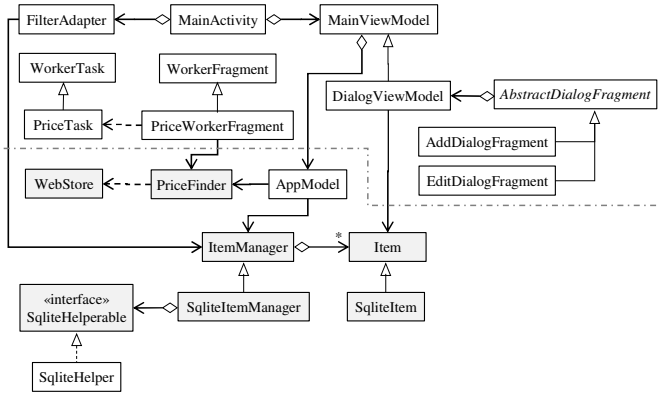

Figure 3. Design of the Android version

IV. ANDROID APP

The class diagram in Figure 3 shows the design of the Android version of the application. It also uses MVC, the most popular architectural design for Android apps [2] [7]. The structure of model classes shown in the bottom half of the diagram is identical to that of the Java version. The business logic of the application – managing items, scaping their prices from Web documents, and persisting items to a database – is identical regardless of the platforms. The detailed design and code are also reused from the Java implementation (see Section III). In particular, the classes shown in shaded boxes are identical to those of the Java version except for minor differences in some classes. The Webstore class, for example, represents a Web store icon differently in the two versions, i.e., as an image icon in Java and as an app resource identifier on Android. It is possible to make the code more reusable by factoring out common code into a base enum type and introducing platform-specific extended enum types [1]. The SQliteHelper class is, of course, rewritten by using the SQLite API included in the Android SDK. The AppModel class is a new model class introduced specifically for the Android version. It represents the app state that has to be retained even if the app restarts because of configuration changes such as screen orientation changes (see below for more on this).

The interesting part is the design of user interface-related classes shown in the top half of the diagram as it is different from that of the Java version for several reasons. In Android, the user interface layout – composition of views and widgets – are declared in XML files called *layout resources*, thus the user interface classes shown in the diagram are control code or view-specific model classes. That is, unlike the Java version, there are no widget classes. Remember also that an activity is an Android application component that represents a single screen with a user interface. The primary roles of an activity class are to find views and widgets, set their properties, and register event handlers for them. The MainActivity class, for example, is the activity responsible for the main screen that displays items in a list view, and the FilterAdapter provides the list view with the items to be displayed based on the user's selection by adapting the ItemManager from the model part. There are other activities and view-specific model classes suppressed in the diagram.

The user interface design shows the use of Android-specific concepts and framework classes, not present in the Java version: view models, fragments, and tasks. You can see classes named as MainViewModel and DialogViewModdel that work as middlemen, or *data binders*, between the view and the model. These so-called *view model* classes of the Android store and manage user interface-related data in an activity lifecycle conscious way [2] [8]. An application component such as an activity may be paused, stopped, destroyed, and recreated by the system. For example, when the screen is rotated, the running activity destroyed. And a new instance is created with a different configuration, e.g., a landscape layout. Android provides several ways to let application data survive configuration changes. One simple and stable way is to use a view model for persisting the user interface. A view model object is automatically retained during configuration changes, and the data it holds is immediately available to the new activity created due to the configuration change. The newly created activity becomes the new owner of the view model. The MainViewModel class is a view model for the main screen, and its subclass DialogViewModel class is for various dialogs.

Besides view models, the user interface design uses another Android-specific concept called fragments. A *fragment* is a modular section of an activity with its own lifecycle. It is a piece of a user interface or behavior that can be placed within an activity. For example, an activity can consist of multiple fragments to provide a multi-pane user interface. In the design, fragments are used for two different purposes. They are used to code several dialogs as shown on the top right side of the class diagram. More interestingly, fragments are used to preserve network operations during configuration changes such as screen orientation. The Android system can retain an instance of a fragment when an activity is recreated due to a configuration change. A fragment, therefore, can be used to retain active objects such as threads, asynchronous tasks, and network tasks across activity instances. In the design, a network operation such as scraping Web documents is wrapped in and managed by a retained fragment, called a *worker fragment.* A worker fragment

plays a similar role as a view model but for computation. It preserves a computation or operation during a configuration change. The Android system prohibits a network operation on the main, or user interface, thread to prevent an unresponsive user interface. All network operations, therefore, have to be performed asynchronously on a thread other than the main thread. The class WorkerTask was introduced for this purpose, i.e., to perform a network operation asynchronously, and is used by the WorkerFragment class.

Table 2. Code size of the Android version shows the size complexity of the Android version of the application in terms of classes and source lines of code (SLOC). The Android implementation has 29 classes and 3747 SLOC.

Table 2. Code size of the Android version

Classes	**No. of Classes**	*Percent (%)*	**No. of Lines**	*Percent (%)*
UI	19	65.52	2567	68.51
Data	3	10.34	386	10.30
Storage	4	13.79	248	6.62
Network	3	10.34	546	14.57
Total	29	100	3747	100

V. Evaluation

As shown in the previous two sections, the Java version consists of 22 classes with 3157 SLOC and the Android version 29 classes with 3747 SLOC. In terms of SLOC, the Android version is 19% bigger than the Java version. In both versions, most code is concerned with user interfaces (UI): 68% for Java and 69% for Android. The application is UI-intensive in that you write more UI-related code than the core business logic. We wrote 20% more UI-related code for Android (2567 SLOC) than Java (2145 SLOC). This is very surprising because the Android user interfaces are declared in XML and provided as layout resources whereas the Java user interfaces are coded in Java. There are nine user interface resource files written in XML, including layouts for activities and dialogs, menus for app bars and popups, and settings. There are no view or widget classes coded in Java. Our initial expectation was that the UI-as-code approach of Java should require more coding than the declarative user interface approach, or UI-as-XML, of Android. In the Java version, all UI-related classes except for the ItemListModel class are indeed user-defined widget classes, representing different parts of the UI including dialogs and menus. They do what the Android user interface classes plus the XML layout files do.

Why does the Android version require more UI-related code than Java even with its user interface layouts defined in XML? To find this out, we examined our Android source code. We first grouped our classes by their roles and measured their code sizes as shown in Table 3. The first two groups – classes responsible for composing and showing the main screen and different dialogs – account for 62% of the UI code. Even with UI layouts such as screens, dialogs, and menus defined in XML, a non-trivial amount of code was written to compose and customize them at runtime. These classes are also responsible for handling screen orientation changes (see below for a discussion on this). Two classes adapt business logic classes to supply data, i.e., items being watched, to the main screen, and they account for 16% of the UI code. The group labeled "config. changes" is interesting because its classes are Android-specific and don't appear in the Java version. This group includes classes like view models and worker fragments that are responsible for handling configuration changes such as screen orientation, and it accounts for 16% of the UI-related code.

Table 3. Size of UI-related code

Group	**No. of Classes**	*Percent (%)*	**No. of Lines**	*Percent (%)*
Main screen	3	16.67	959	37.36
Dialogs	5	27.78	625	24.35
Model adapters	2	11.11	399	15.54
Config. changes	6	33.33	405	15.78
Miscellaneous	2	11.11	179	6.97
Total	18	100.00	2567	100.00

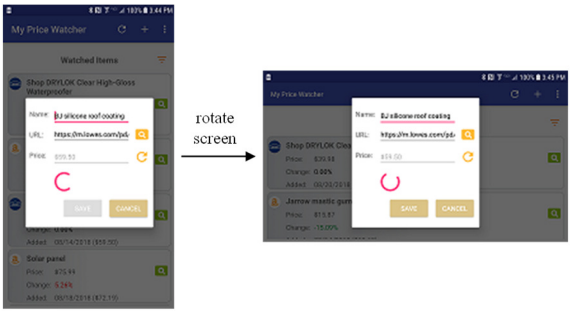

Figure 4. Screen orientation change

When a configuration change such as screen orientation occurs during the runtime, Android restarts an activity by destroying the current instance and creating a new one. This restart behavior is an Android way of adapting to a new configuration by automatically reloading the app with alternative resources that match the new configuration. This means that the app code is responsible for preserving the app data as well as computation during a configuration change. For example, if the user rotates the screen while adding a new item to the watch list (see Figure 4), the Android system will pause, stop, and destroy the running activity and then restart it by creating a new instance possibly with a landscape layout. The app code is responsible for showing the same contents as before with the new user interface. The app data has to be saved and restored, and the computation, e.g., the network operation to find the initial price of the item being added, has to be preserved. As shown in the previous section, we used view models and worker fragments for this. We closely examined the classes in the first two groups (Main screen and dialogs) and 14% of their code (221 SLOC) is for addressing configuration changes. We, therefore, have a total of 626 SLOC for addressing configuration changes including six standalone classes with 405 SLOC and 221 SLOC embedded in other classes. And the device configuration changes such as screen orientation account for 24% (626 / 2567 * 100) of the UI-related code and 17% (626 / 3747) of the overall code.

Besides the roles of code shown in Table 3 above, there is code that is not written as a separate class but scattered over the UI-related classes. The code that we are interested in is the one that is Android platform-specific and thus doesn't appear in the

Java version. This is code other than the so-called *control code* that handles user interactions.

- Android provides platform-specific programming and framework concepts such as activities, fragments, and intents. Coding these can be somewhat involved because they are not provided as built-in language constructs or features. Unlike Java, for example, navigating between screens (activities) isn't as simple as calling a method, as an activity may run in a different process and virtual machine. Parameters, if any, have to be manually packed and unpacked with no parameter checking done by the framework. Return values require the use of a callback mechanism – i.e., overriding a specific framework method. The resulting code is not only less reliable but also requires more work.

- A mobile user interface has to be responsive as an unresponsive user interface makes the device unusable. Android prohibits network operations on the main (user interface) thread to avoid creating an unresponsive user interface. All network operations have to be performed asynchronously on a thread other than the main thread. As described in Section IV, all network operations in our code are performed by special classes called worker tasks. Performing network operations asynchronous is of course a good practice to improve the responsiveness of an app, however, it requires additional code for creating and managing threads. And it also introduces complications due to multithread programming such as mixing asynchronous and synchronous code.

- Because of the mobility of a device, a mobile app needs to check the current situation constantly and adapt to changes, e.g., availability and strength of WiFi and GPS signals. Since our app uses Internet resources, we check the availability of a network connection but just once when the app is first launched. In practice, however, you may need to monitor the network connection while the app is running. If a connection is not available, your app may try to establish a new connection automatically or show the built-in connection settings app. Establishing a connection automatically would be ideal but might be involved because of reasons such as sign-in requirements or mobile data costs. All these, of course, mean additional coding in addition to the business logic.

Security and privacy can be another type of concern that is common and has to be addressed in the Android code. Android uses permission-based access control to protect its resources. If an app requires a group of permission called *dangerous permissions* – permissions that could affect the user's privacy or the device's normal operation, e.g., contacts and SMS – the user must explicitly grant these permissions at runtime. This of course requires additional code even if it may be boiler-plate code. There seem to be many such secondary concerns due to the nature of mobile platforms, e.g., diversity and mobility. It would be interesting future work to identify and classify them and study their impacts on source code complexity.

VI. Conclusion

Mobile apps run on small devices such as smartphones and tablets, and they are smaller than traditional applications. These and other aspects of mobile apps may have given a wrong impression to beginning programmers as well as the general public that writing mobile apps are simpler than traditional applications. We showed through a small experiment that Android mobile apps written in Java require more source lines of code (SLOC) than equivalent Java desktop applications. The sample application coded in our experiment consists of 22 classes with 3157 SLOC for the Java version and 29 classes with 3747 SLOC for the Android version. The Android version is 19% bigger than the Java version in source code size. This size difference was observed mainly in the user interface (UI)-related code as the business logic of the application is the same regardless of the platforms. In fact, the business logic code was reused, or shared, between the two versions. We wrote 20% more UI-related code for Android (2567 SLOC) than Java (2145 SLOC). This is quite surprising because the Android user interfaces are declared in XML whereas the Java user interfaces are coded in Java. Android's UI-as-XML approach requires more code than the UI-as-code approach of Java. The coding overhead of Android, however, is mostly due to the requirements and constraints specific to mobile platforms, e.g., diversity, mobility, responsiveness, security, and privacy. For example, the device configuration changes such as screen orientation account for 24% of the UI-related code and 17% of the overall code. We also found that the Android code is a lot more involved and complicated because of code tangling and scattering. The core user interface logic – displaying data – is mixed with additional code for platform-specific requirements, spread over multiple program modules.

References

[1] Y. Cheon, C. Chavez and U. Castro, Code reuse between Java and Android applications, Proceedings of the 14th International Conference on Software Technologies, Prague, Czech, July 26-28, 2019, pages 246-253.

[2] A. Daoudi, et al., An exploratory study of MVC-based architectural pattersn in Android apps, *ACM/SIGAPP Symposium on Applied Computing*, April 2020, pp. 1711-1720.

[3] U. A. Mannan, et al., Understanding code smells in Android applications, *IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, Austin, TX, 2016, pp. 225-236.

[4] P. Minelli and M. Lanza, Software analytics for mobile applications – insights & lessons learned, *European Conference on Software Maintenance and Reengineering*, Genova, Itally, 2013, pp. 144–153.

[5] *Number of apps available in leading app stores as of 1st quarter 2020*, accessed 24, 2020, https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/.

[6] M. D. Syer, et al, Revisiting prior empirical findings for mobile apps: An empirical case study on the 15 most popular open-source Android apps, *Conference of the Center for Advanced Studies on Collaborative Research*, Riverton, NJ, 2013, pp. 283–297.

[7] R. Verdecchia, I. Malavolta and P. Lago, Guidelines for architecting Android apps: A mixed-method empirical study, *IEEE International Conference on Software Architecture*, Hamburg, Germany, 2019, pp. 141-150.

[8] *ViewModel Overview*, accessed 24 August 2020, https://developer.android.com/topic/libraries/architecture/viewmodel, pp. 1-8.