Code Reuse Between Java and Android Applications

Yoonsik Cheon, Carlos V. Chavez and Ubaldo Castro

TR #19-50
May 2019

Keywords: code reuse, multiplatform application, platform difference, software development, Android, Java

2012 ACM CCS: • Software and its engineering ~ Software design engineering • Software and its engineering ~ Agile software development • Software and its engineering ~ Reusability • Information systems ~ Mobile information processing systems • Human-centered computing ~ Mobile computing

An extended version of the paper to be presented at the *14th International Conference on Software Technologies (ICSOFT 2019)*, Prague, Czech Republic, July 26–28, 2019.

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

Code Reuse Between Java and Android Applications

Yoonsik Cheon, Carlos V. Chavez and Ubaldo Castro
Department of Computer Science, The University of Texas at El Paso, El Paso, Texas, U.S.A.
ycheon@utep.edu, {cvchavez2, ucastro}@miners.utep.edu

Keywords: Code reuse, multiplatform application, platform difference, Android, Java

Abstract: Java and Android applications can be written in the same programming language. Thus, it is natural to ask how much code can be shared between them. In this paper, we perform a case study to measure quantitatively the amount of code that can be shared and reused for a multiplatform application running on the Java platform and the Android platform. We first configure a development environment consisting of platform-specific tools and supporting continuous integration. We then propose a general architecture for a multiplatform application under a guiding design principle of having clearly defined interfaces and employing loose coupling to accommodate platform differences and variations. Specifically, we separate our application into two parts, a platform-independent part (PIP) and a platform-dependent part (PDP), and share the PIP between platform-specific versions. Our key finding is that 37%–40% of code can be shared and reused between the Java and the Android versions of our application. Interestingly, the Android version requires 8% more code than Java due to platform-specific constraints and concerns. We also learned that the quality of an application can be improved dramatically through multiplatform development.

1 INTRODUCTION

Java is one of the most popular programming languages in use today for developing a wide spectrum of applications running on a range of platforms from mobile and desktop to server (TIOBE, 2019). The Android operating system is the dominating mobile platform of today (IDC, 2019), and Android applications can be written in Java albeit some differences between the Java application programming interface (API) and the Android API. Source code reuse is considered as a fundamental part of software development (Abdalkareem et al., 2017). Thus, it is natural to ask how much code can be shared and reused between Java and Android applications. In this paper, we study and answer this question.

Our research question is: how much code can be shared and reused in developing multiplatform applications written in the same programming language? A *multiplatform application* is an application that is developed for and runs on multiple platforms. In this paper, we use the term platform loosely to mean operating systems, runtimes including virtual machines, and software development kits (SDKs). We include SDK in the platform, as our research is focused on code reuse and we are also interested in studying the impact and complications caused by API differences of SDKs. The two specific platforms that we consider in this paper are Java SDK for desktop applications and Android Java SDK. At the time of writing, Java is the primary language for developing Android applications; Android tools also support other programming languages such as Kotlin and C/C++.

We answer the above research question by performing an experiment to measure quantitatively the degree of code reuse possible between Java and Android versions of an application. Our experiment is a case study developing a Java application and its Android version running on mobile devices like smartphones and tablets. The application is small in Java but is a typical Android application of an average size (Minelli and Lanza, 2013), requiring a graphical user interface (GUI), data persistence, networking, and a bit of multithreading. The application helps a user to figure out the best time to purchase products by watching over fluctuating online prices (see Section 2). We develop the application incrementally and iteratively with continuous integration and testing by switching instantly between platform-specific integrated development environments (IDEs). For this, we configure a custom multiplatform application development environment by sort of gluing individual platform-specific tools and IDEs. It propagates immediately changes made on the shared code

using one IDE to other IDEs and platforms. We attempt to maximize code reuse between two versions of the application. Our design approach is to decompose an application into two different parts: a *platform-independent part* (*PIP*) and a *platform-dependent part* (*PDP*). The PIP is the part of an application that does not depend on platform specifics and thus can be reused on different platforms. The guiding design principle is to have clearly defined interfaces and employ loose coupling between the two parts to accommodate platform differences and variations. To determine code reuse, we measure the size of our code with a simple metric counting the number of lines of code (LOC). We also share other findings and lessons that we learned from our case study.

One difference of Android from Java is that its devices are resource-constrained in storage capacity and battery lifetime, and thus memory efficiency is an important quality factor for Android applications (Cheon et al., 2017) (Sillars, 2015). The performance constraints may affect the design and coding of an application and thus its code reuse. In this paper, however, we don not consider the performance in the implementation of the Android version of our application.

The rest of this paper is structured as follows. In Section 2, we describe our case study application, including the third party libraries and frameworks we use for its implementation. In Section 3, we explain our development of the application, starting with the configuration of tools and some design challenges along with our solutions. In Section 4, we assess our development in terms of code reuse by measuring our code quantitatively and interpreting the measurements. In Section 5, we mention few related work, and we conclude this paper in Section 6.

2 CASE STUDY

As mentioned in the previous section, we perform a small but realistic case study to find an answer to our research question on multiplatform code reuse. The primary objective of our case study thus is to measure quantitatively the amount of code reuse possible between a Java application and its Android version. The secondary objective is to learn any complications or issues associated with multiplatform application development. They may be related with not only the design and coding of an application but also configuration of tools for incremental application development with continuous integration.

In our case study, we develop a Java desktop application and its Android mobile version, named Price Watcher. The application tracks the prices of products, or items, extracted from their webpages (see Figure 1 for sample screenshots). The application helps a user to figure out the best time to buy items by watching over fluctuating prices. As the prices are scraped from webpages, the watch list may consist of items from different online stores or websites.

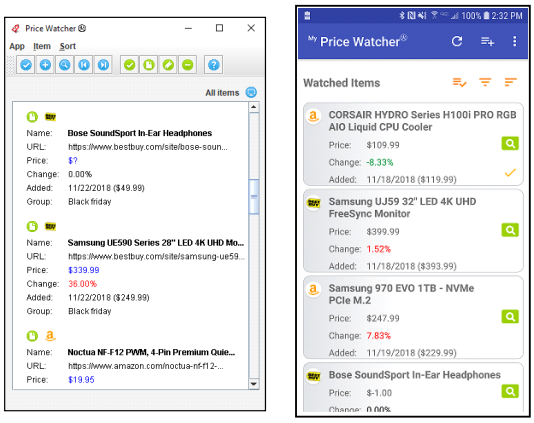

Figure 1: Screenshots of Price Watcher (Java and Android)

Most applications of today are built with libraries or frameworks that are either provided by the platforms themselves or acquired from third parties. These libraries and frameworks may introduce additional complications to the development of multiplatform applications. To study their implications in the code reuse, we use several different libraries and frameworks in our development. The APIs of Java and Android Java are very similar for common libraries such as collections, file I/O, and networking. However, graphical user interface (GUI) frameworks of Java and Android are completely different, as Android offers its own framework for GUI programming. Interestingly, there are also differences in the ways third-party libraries and framework are provided or supported by platforms.

- Plain old Java objects (POJOs). A library or framework is written in the ordinary Java, not bound by any special restriction other than those of the Java language specification. It is often bundled in the SDK. For instance, Android SDK includes one particular open source JSON library while Java SDK does not. An open-source library or framework can also be tightly integrated into the platform and be part of the platform's APIs. That is, it is provided as a built-in feature of the platform, often modified significantly through the integration. A good example is Android's support for SQLite, a lightweight, serverless relational database system (SQLite Consortium, 2019). There are noticeable differences be-

tween the Android-specific SQLite API and the JDBC-based Java SQLite API.

- Platform-specific SDK. A third party framework is often provided as a platform specific SDK. An example relevant to our case study is Google's Firebase Database, a cloud-hosted database (Google LLC, 2019). Google provides different Firebase SDKs for different platforms, and the APIs of Android and Java (Firebase Admin Java SDK) are similar but with some subtle syntactic and semantic differences.

We will use three different approaches in our case study to persist the items being watched: a local file (JSON), a SQLite database, and a Firebase cloud storage. As said above, the main reason for doing this is to study their implications in the code reuse.

3 DEVELOPMENT

We develop our application incrementally—one feature at a time—and iteratively by continuously integrating and testing code written on different platforms (Cheon, 2019). In this section, we first show how to configure a multiplatform development environment supporting our development approach. The key requirement is to immediately propagate changes made on the shared code to other IDEs and platforms so that we can switch between IDEs and platforms for simultaneous work on multiple projects. We then point out some of the design challenges associated with multiplatform applications and describe our solutions. Our primary design goal is to maximize code reuse between two versions of our application, and toward that goal we propose a general architecture for multiplatform applications that recognizes and separates platform specifics. Our guiding design principle is to have clearly defined interfaces and employ loose coupling to accommodate platform differences and variations.

3.1 Tools

It is crucial to have adequate tool support for multiplatform application development. Since we develop two versions of an application, one for each platform, our development environment logically consists of three IDEs: two platform-specific IDEs and one for developing the common, sharable code. For Android, we of course use Android Studio, the official IDE from Google for Android application development built on JetBrain's Java IDE called IntelliJ IDEA. For the development of both the Java application and the common code, any reasonable Java IDE including IntelliJ IDEA will work. However, since we are also interested in learning complexities and issues caused by diversity of tools, we opt for Eclipse. It is very likely that a combination of several tools be used for multiplatform application development. We configure our own multiplatform application development environment by composing a few platform-specific tools or IDE as suggested in (Speicher and Cheon, 2018).

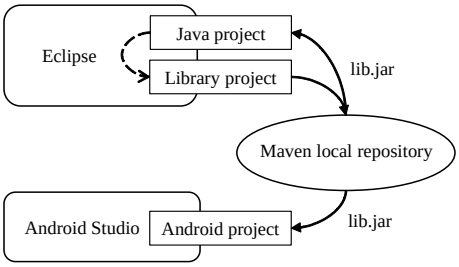

Figure 2: Development environment

Figure 2 shows our development environment consisting of Eclipse and Android Studio. We develop our application incrementally and iteratively with continuous integration and testing. One key requirement for incremental development of a multiplatform application is to propagate changes immediately from one IDE to the other. We use Apache Maven (Apache Software Foundation, 2019), a software project management and build tool, to share code and propagate changes in the form of a library, a Java archive (jar) file. Both the Java project and the Library project in Eclipse are Maven projects, and the Library project produces a library jar file, called an *artifact* in Maven, of the common code. The library jar file is installed in the Maven local repository and becomes available instantaneously to the Android project. Android Studio uses the Gradle build tool that understands Maven repositories. For the Java project we can also make it reference, or depend on, the Library project by changing its build path, as both are Eclipse projects. This allows us to also use Eclipse to build the Java project.

3.2 Design

Our primary design goal is to maximize code reuse between Java and Android versions of the application. For this, we decompose our application into two parts: a platform-independent part and a platform-dependent part. The *platform-independent part* (*PIP*) is the part of an application that does not depend on specifics of implementation platforms like platform-specific APIs (Cheon, 2019). In the model-view-controller (MVC) architecture, the model is a good

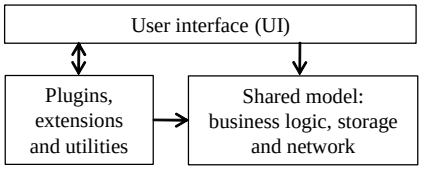

Figure 3: Application architecture

candidate for the PIP. The *platform-dependent part* (*PDP*) is the part of an application that does depend on a specific platform, e.g., the view and view-specific controller of MVC (Cheon, 2019). We make this distinction to share the PIP code across platforms while developing a specific PDP on each target platform. Thus, the key criterion on identifying and determining the PIP of an application is whether the code can be shared on all the target platforms of the application. Our notions of PIP and PDP are similar to those of platform-independent models (PIM) and platform-specific models (PSM) in model-driven software development (Brown, 2004) (Meservy and Fenstermacher, 2005). As said earlier, Maven plays a central role in our development by immediately propagating changes made on the PIP to other tools and IDEs.

One challenge in the design and coding of our application is to cleanly separate the PIP from the PDP while minimizing code duplication. It is easy to say and difficult to do. We need to identify commonalities and differences between the two target platforms in terms of the APIs and libraries needed for the implementations of the application. The PIP—the common, shared code—has to accommodate the platform differences and variations. Figure 3 shows the high level design of our application. The box labeled "Shared model" is the PIP and the other two are the PDP. Since Android provides its own GUI framework along with Android-specific concepts such as activities and fragments, the biggest platform difference is the UI. The other is to encapsulate platform differences and variations for the PIP (more on this below and later sections). As expected, the PIP does not depend on the PDP.

How does the PIP accommodate platform differences and variations? In a single platform application, the PIP and the PDP code are often interwoven and tangled. For a multiplatform application, we need to separate them and make the dependencies between them clean and explicit. In particular, we need to eliminate any dependency of the PIP on the PDP. There are several different techniques and approaches possible, such as required interface, inheritance and hook, parameterization, interface cloning, and interface unification (Cheon, 2019).

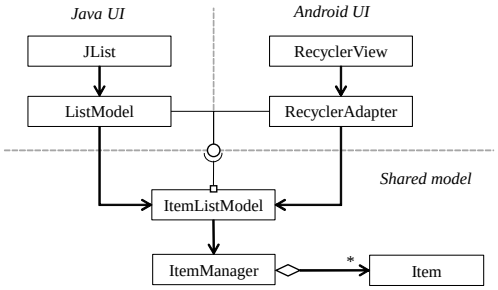

Figure 4: Plugin-based approach

3.2.1 Plug-ins via Required Interfaces

The implementation of the PIP often depends on that of the PDP. As an example, consider the case when a new item is added to the watch list through the UI or externally through the shared cloud storage. The notions of items and the watch list can be coded once in the PIP and thus the PIP will be responsible for adding the item to the watch list. The PIP however cannot display the newly added item. Instead, it has to tell the UI (PDP) to display the newly added item, and each platform provides a different way of telling the UI to update its display. We eliminate this kind of dependencies by applying the *dependency inversion principle,* a specific form of decoupling program modules (Martin, 2003). We let the PIP depend on an abstraction of the PDP, not its concrete implementation. The assumption that the PIP makes on the PDP or its environment is coded explicitly in the form of a *required interface* (F. Bronsard, et al., 1991), an interface that is defined by a service provider of an interaction that specifies what a service consumer or client needs to do so that it can be used in that interaction. In a sense, this approach allows one to plug in platform-specific code to the PIP by providing a class that implements the required interface.

Figure 4 shows an application of this approach. A model class named ItemListModel manages the items that are currently displayed by the UI, and it defines a required interface to interact with a platform-specific UI. Both the Java and the Android applications provide an implementation of the required interface coded using platform-specific APIs. In the figure these are shown using the socket/lollipop notation. In a sense, the platform-specific UI is a plug-in that can be inserted into the PIP. The class diagram also shows how we maximize code reuse and minimize code duplication. The UI needs to display the items being watched, and each platform of course provides a different view, or widget, for displaying a collection of data, e.g., JList in Java and RecyclerView in Android. This so-called container view stores the data

differently in a view-specific model classes, e.g., ListModel and RecyclerAdapter, providing a different set of operations. However, there is enough commonality in the set of operations, including grouping, filtering, sorting, and searching, that need be implemented for our application. Therefore, our design decision is to introduce a platform-neutral model class, ItemListModel, in the PIP and let platform-specific model classes in the PDP to delegate their operations to it.

3.2.2 Templates via Inheritance

Another way to accommodate platform differences and variations is to provide a skeleton algorithm, or code, in the PIP and let the PDP fill out the details in a platform-specific way. For this we use well-known software design patterns such as template method, strategy, factory method, and abstract factory (Gamma et al., 1994). This approach gives less freedom to the PDP than the previous approach, as the PDP has to follow the skeleton algorithm defined in the PIP, but more code is shared and reused. Figure 5 shows an application of this approach, our design for storing the items in a cloud storage from Google called Firebase (Google LLC, 2019). We define a special item manager class to store all items in a cloud storage. The idea is to override all mutation methods inherited from its superclass to make the items persistent by storing them in a cloud storage. As mentioned in the previous section, Firebase is supported in several different platforms including Android and Java. However, Google provides different Firebase SDKs for different platforms, and the APIs of Android and Java (Firebase Admin Java SDK) have subtle differences (Google LLC, 2019). In a Java application, for example, one has to write code to authenticate and initialize the application to use Firebase, which on Android is taken care of by the system. We encapsulate this platform difference in a separate, abstract class named FireBaseHelper and let the PDP complete it through subclassing. The abstract class defines a skeleton algorithm, or a sequence of steps, for using a Firebase database in terms of hook methods that can be overridden by a platform-specific subclass like the JavaFireBaseHelper shown in the figure. A similar approach is used to persist items using a local file and a SQLite database.

3.2.3 Building Blocks via Classes

Instead of providing a set of well-defined extension points in the form of pluggable interfaces or abstract classes with hook methods as described above, we can provide platform-neutral complete classes to be used as building blocks, or components, by the PDP. The

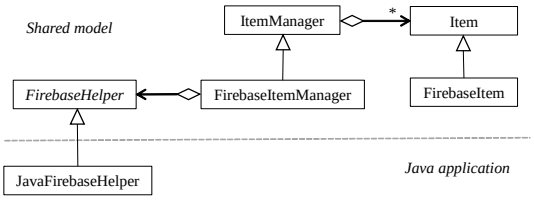

Figure 5: Template-based approach

PIP provides minimal support for PDP, but the PDP has full control over the use of the provided components. This approach can be a good alternative if there are big API differences among the target platforms or it is hard to come up with a well-designed, unified API for all platforms. In our application, for example, the list of supported online stores, or websites, can be different on each platform, and the website list can be used in several platform-specific ways, e.g., creating UI elements such as menu and action items. Some website attributes like icons are also stored and manipulated in platform-specific ways. We thus decide to define an enum type in the PIP that gives all websites that are supported by both platforms. Each website of the enum type defines only common, platform neutral attributes and operations, including its name, URL, and a command object to find the price of an item from its webpage. The PDP defines a platform-specific enum type by adding platform-specific attributes like website icons to the enum values provided by the PIP and, if any, adding platform-specific websites.

There are also several other techniques that can be used to address platform differences, and some of which will be mentioned and described in Section 4.3.

3.3 Implementation

We performed iterative and incremental development in our case study for two different reasons. First, we were not familiar with multiplatform application development and thus wanted to take advantage of what we learned during the development of earlier increments or versions of our application. Second, we wanted to refine our experiment as the development progresses by adding new application features partly based on the preliminary findings from the early increments. This was the exact reason that we ended up supporting three different persistent storages (file, database, and cloud storage) even if only one is needed for our application. Our initial plan was to store item data in a local database.

For each increment we need to work on three different projects: two Java projects in Eclipse and one Android project in Android Studio. The way our

Table 1: PIP classes and their sizes

Class	No. of Lines
Item	150
ItemListModel	295
ItemManager	246
FileHelper	12
FileItem	45
FileItemManager	106
SqliteDatabaseHelperable	19
SqliteItem	28
SqliteItemManager	114
FirebaseHelper	137
FirebaseItem	93
FirebaseItemManager	129
PriceFinder	35
WebPriceFinder	64
WebStoreBase	317
Log	35
Total	1825

development environment is configured allows us to work on all three projects at the same time by moving the mouse back and forth between Eclipse and Android Studio. However, it is more productive to first develop an application for one platform, create or derive a PIP from the application by refactoring its code, and then refine the PIP by writing the application for the other platform. The reason is that the PDP captures what is specific or unique on each platform, and the PIP is a generalization of all platforms. The PIP is coded by identifying and specifying algorithms and data that are common on all platforms. The complete Java application consists of 36 classes and 4604 lines of source code, and the Android version consists of 41 classes and 4987 lines of source code. In the next section we will measure our code to analyze code reuse between the two versions.

4 ASSESSMENT

In this section we first measure the size of our code with a simple metric counting the number of lines of code (LOC) including comments and blank lines that our classes take up in the source code files. We then analyze and interpret the measured LOCs. We also describe other findings and lessons that we learned from our development.

4.1 PIP

The PIP consists of 16 Java classes and 1825 lines of source code (see Table 1). Nested classes and interfaces are not included in the number of classes, but their code were of course counted in the number of

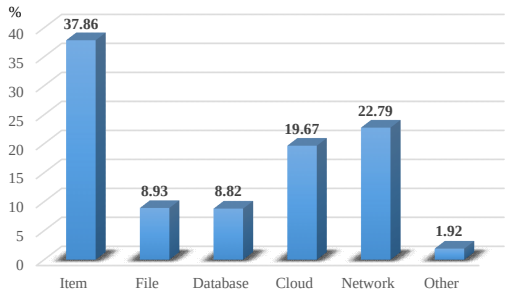

Figure 6: Distribution of PIP code

source code lines. We can group the PIP classes into four different categories for use later in our analysis.

- Items: Classes in this group implement the core business logic of our application and include Item, ItemManager and ItemListModel. They store information about the items being watched and manage the watch list by providing operations such as adding, removing, updating, filtering, and sorting.
- Storage: As mentioned in the previous sections, we support three different approaches to persist item data: file, database, and cloud storage. For each data persistence approach, we provide three classes: a manager as a subclass of ItemManager, a helper to encapsulate platform differences of the storage APIs, and a special item class as a subclass of Item to store storage-specific item information such as a database id or cloud key.
- Network: These are network-related classes including classes for online stores, or websites, and for finding an item's price from its webpage. We define a price finder class for each online store.
- Others: Utility and miscellaneous classes.

It would be very interesting to see how the code is distributed among these groups of classes (see Figure 6). The item-related classes account for 38% of our code, the storage 38%, and the network 23%. If we look further into the items group, 43% (295 lines) of source code belong to the ItemListModel class. Remember this is a UI-specific model class that we introduced to maximize code reuse (see Section 3.2). The class helps the UI to display watched items in many different ways; it is not really a core business logic class. Therefore, we can infer that our application is UI-intensive. Another thing we can quickly learn from the graph is that the three data persistence approaches have varying code sizes: file (9%), database (9%), and cloud (20%). This may indicate either the complexities of the approach themselves or the degrees of their code reuse possible. We will discuss

Table 2: Sizes of application code

App	Part	No. of Classes	No. of Lines	Percent (%)
Java	PIP	16	1825	40
	PDP	20	2779	60
	All	36	4604	100
Android	PIP	16	1825	37
	PDP	25	3162	63
	All	41	4987	100

more on this later in this section.

4.2 PDP

Table 2 shows the sizes of both the Java and the Android versions of our application including their PDPs. The source code lines of the Android PDP include only manually-written Java code; they do not include so-called resource files such as GUI layouts written in XML or automatically generated Java source code files. The last column of the table shows the percentage that each part accounts for the whole application code, calculated using a formula, *x* / (PIP + PDP) * 100, where *x* is either PIP or PDP. The PIP takes 40% and 37% of the Java and the Android application code, respectively. That is, 37%–40% of code are reused in our application despite being UI-intensive. One side finding is that the Android PDP requires 14% ((3162 - 2779) / 2779 * 100) more code than Java. This may be partly because the Android version provides a few additional features. However, Android applications generally require more coding to address Android-specific concerns such as application lifecycles and screen orientation changes. The overall size of the Android version is 8% larger than the Java version.

How many lines of code are needed to interface with the PIP? Table 3 lists platform-specific classes along with the numbers of source code lines written to interface with the PIP. Note that some of the classes are named the same as those of the PIP, but they are platform-specific subclasses defined in the PDPs. The last row shows the percentages of the interfacing code in the PDPs, 14% for Java and 22% for Android. As mentioned before, Android requires more coding, and this is clearly shown in the table. Two bulky classes are RecyclerAdapter and WebStore. The first class addresses Android-specific UI concern, e.g., recycling widgets to display multiple items. The additional code of the second class is mainly due to one more online store supported in the Android version; Android-specific features were used for this. Another thing to notice is the relative amount of code for three data persistence approaches. The database approach

Table 3: PDP code for interfacing with PIP

Class	No. of Lines	
	Java	Android
ItemListModel	78	N/A
RecyclerAdpater	N/A	237
FileHelper	32	38
SqliteDatabaseHelper	187	117
SqliteItemManager	N/A	20
FirebaseHelper	39	22
WebStore	47	90
WebPriceFinder	N/A	36
Total	383	560
Percent (%, total/PDP)	14	18

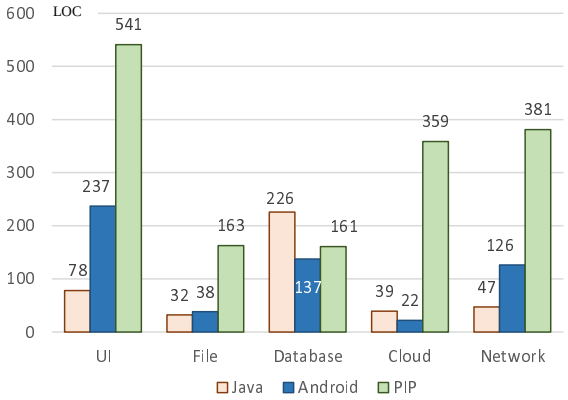

Figure 7: PIP features and their interfacing code in the numbers of lines

requires 3–6 times more code (see below for more discussion on this).

It would be very interesting and instructive to see how the platform differences of the same API affect reuse of the PIP code. If we combine the data from Table 1 and Table 3, we can estimate the amount of code that has to be written to reuse the major features of the PIP. Figure 7 shows this by plotting the sizes of the PIP classes and the corresponding PDP code for three features: UI, storage and network. The PDP code of course includes only the interfacing code—code written to interface with the PIP feature in question. Figure 8 shows the same information but in percentages. As shown, the percentages vary widely from 6% (of Android cloud) to 58% (of Java database) among the PIP features; the average is 18% for Java and 25% for Android. Th LOC numbers in Figure 7 generally indicate the degrees of code reusability as well as the easiness of reuse. For example, the first bar says one needs to write only 78 lines of Java code instead 619 (= 78 + 541) lines to manage watched items and help the UI to display them. For each PIP feature the size of its interfacing code is marginal compared to coding the whole logic in the PDP.

However, there is one exception. Among the

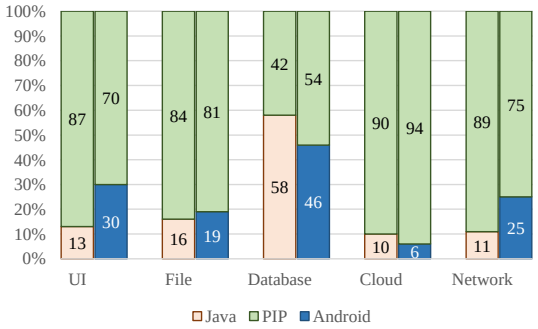

Figure 8: PIP and interfacing code in percentages

Table 4: Complexity of data persistence in LOC

Storage	Java	Android	Average
File	195	201	198
Database	387	298	343
Cloud	398	381	390

three data persistence approaches of file, database and cloud storage, the database approach requires more code in the PDP side—58% for Java and 46% for Android (see Figure 8). Before we look into this unusual case, let us first measure the complexity of each of the persistence approach in terms of source code lines. As shown in Table 4, the file-based approach is smaller than the database that requires somewhat less code than the cloud storage.

One thing that the table does not show is the similarity of the platform APIs between Java and Android. The Android File (I/O) APIs are exactly the same as those of Java except for additional notations for denoting different kinds of storages (internal and external) and directories (download, documents, etc.). The Google's Firebase APIs for Java and Android are also very similar albeit some subtle differences in some of the operations provided. However, the story is completely different for the SQLite database. Android supports it as sort of a built-in feature with an API tightly integrated with its own framework. For Java, there is an open-source, JDBC-based library for SQLite databases. Due to this difference of platform APIs, we had to push more code to the PDPs, and that is why more code was written in the PDPs. In short, the platform APIs and their commonalities and differences greatly affect the development of the PIP and its reusability.

How does the separation of PIP and PDP affect the size of an application? Is there an increase in the code size caused by the separation, and if so, how much? The way we develop our application allows us to answer this question with minimal effort. We

Table 5: Overheads of PIP and PDP separation

App	Part	No. of classes	No. of lines	Overhead (%)
Java	PIP + PDP	36	4604	14
	Monolithic	27	4031	
Android	PIP + PDP	40	4987	13
	Monolithic	31	4418	

wrote our application incrementally by writing complete code on one platform for the feature under development and then refactored the code to derive a reusable PIP. As a result, we have two versions of our application, written with and without using the PIP. Table 5 shows the sizes of different versions of our application. As guessed, the monolithic versions—the ones written without using the PIP—have less classes and lines of code, and the size overheads of PIP/PDP are 14% and 13% for Java and Android, respectively.

4.3 Other Findings and Lessons

We learned both positive and negative sides of multiplatform application development through our case study. Besides the obvious benefit of code reuse, perhaps, the most important side benefit from a developer's point of view is that it provides opportunities for improving the quality of an application. We have to address diversity of platforms by considering platform-specific restrictions or concerns. In addition we have to work on, or review, the same or derived code several times, each with a different perspective—either as a service provider or a consumer. We first prototyped a new application feature in one of the platforms, refactored the monolithic code into a reusable library (PIP) and the PDP of that platform, and apply the library in the other platform. The implementation of a feature required several iterations with continuous integration and testing on two platforms. This development process allowed us to notice issues and problems from simple mistakes like naming inconsistency to more serious ones.

To be more specific we found that Android-specific features allowed us to explore and test our application in a way that would be impossible or unnecessary for a typical Java application, often exposing a potential issue or problem in the application. An example is device orientation change. The screen on an Android device can switch between portrait and landscape mode in response to the way one holds the device or when the device is rotated. Our application allows the user to filter items to be displayed in several different ways, e.g., based on online stores, item groups and keyword search. An item can belong to a user-named group, but not all items have to belong

to a group. The filtering feature was first introduced to the Java version, and it worked correctly. When the feature was added to the Android version, however, the device orientation change causes the application to show only those items that do not belong to any item group. We soon learned that this strange behavior was caused by an incorrect re-initialization of the application. A re-initialization occurs when the screen orientation changes because Android creates a new instance of a framework class upon screen orientation change; no such re-initialization is needed for the Java version. We fixed the problem by modifying the PIP. We also added a new UI element in both PDPs to provide an option for displaying all those items that do not belong to any item group. This option was overlooked in the initial design of the filtering feature.

Multiplatform application development encourages one to generalize APIs, especially those of the PIP, to address and accommodate platform differences and variations. In fact, even platform-specific restrictions or constraints contribute positively on the creation of a more reusable and extensible application. The initial design of our network module done on the Java platform provided synchronous operations, and a special return value was used to notify when the invoked operations fail. On Android, however, the provided network operations were always called in background threads created by the UI because Android does not allow any network operation on the UI thread (while Java does). This made us to create a new version of the network module that also provides asynchronous operations implemented using the Observer design pattern (Gamma et al., 1994). Due to the use of this design pattern, the error handling was also improved by creating a separate callback method in the observer, or listener, interface. The error reporting is separated from the main logic, and thus more detailed information about the error is provided to the caller. Android's emphasis on application responsiveness also made us to improve the user experience of our application by providing additional features such as setting network timeouts and canceling network operations.

We noticed different kinds of platform differences and variations, including APIs (both syntactic and semantic), literal constant definitions and externalization, and use of external resources such as images. We used a range of techniques to accommodate these platform differences, including required interfaces, callbacks, delegation, parameterization, operation cloning, and design patterns like template methods, factory methods and strategy patterns (see Section 3.2).

One obvious downside of the PIP-PDP separation is that the PIP has to be written using the common denominators, or shared traits, of the both platforms. That is, the platform for the PIP is the intersection of the two target platforms. A common platform feature has to provide the same syntactic interface—operation name and signature, class, and package—as well as the semantics. Otherwise, it cannot be directly used in the PIP implementation and has to be pushed to the PDPs. Let us consider a very simple example. All GUI frameworks provide a way to perform a task on the UI thread, as they are single threaded. However, its syntactic interface may be different, e.g., SwingUtilities.invokeLater(Runnable) in Java and Activity.runOnUiThread(Runnable) in Android.[1] A UI-specific PIP module such as ItemListModel (see Section 3.2) sometimes needs to perform a task on the UI thread, but it cannot use the above methods directly. Our solution was to include it in the required interface of the module and let the PDP provide a platform-specific operation.

It is common practice to use the print statement as a simple and quick tool to trace the flow of execution and print interesting values. This use of the print statement is often referred to as *print debugging*. On Android, however, the recommended style is to use a utility class named Log that offers several static methods.[2] The log messages are fed from an Android device to Android Studio and can be viewed with a tool called LogCat. We addressed this issue of an API mismatch by cloning the Log class on the Java platform, which involves (a) defining a stub class in the PIP with the same syntactic interface as the Android and (b) providing an implementation in the Java PDP written in terms of the Java print statements. Note that the stub class is just for compiling the PIP and is not included in the distribution, the library jar file. We had to use a similar approach when a third party library is provided in the SDK of one platform but not in the other. An example is JSON, which was used to store items in a local file. An open-source JSON library is included in the Android SDK but not in the Java SDK.

Regarding debugging, we found it is a really good idea to include source code files in the snapshot library jar file during the development. Android Studio uses them, instead of the skeleton code decompiled from the bytecode, to show stack trace of an exception, thus providing better support for debugging the

[1] There is also a subtle semantic difference when the method is called on the UI thread.

[2] In fact, the Java coding style of the Android Open Source Project states that the print statements should never be used (https://source.android.com/setup/contribute/code-style).

Android PDP code.

5 RELATED WORK

We found no published work measuring code reuse between Java and Android applications. However, source code reuse in Android applications recently received much attention from researchers. One interesting report is that the practice of software reuse is high among mobile application developers (Mojica et al., 2014). One study even reported that 61% of Android application classes appeared in two or more other applications (Ruiz et al., 2012). Unfortunately, the significant code reuse also indicates illegal cloning of classes, code piracy, or even repackaging of applications (Linares-Vásquez et al., 2014) (Gonzalez et al., 2015). Code reuse also impacts the quality of an application, particularly when code is reused in the copy-and-paste manner from online question-and-answer websites such as Stack Overflow (Abdalkareem et al., 2017). Unlike these existing work, our study investigated code reuse between a mobile application and a desktop application written in the same programming language, where the development processes and practices can be quite different (Minelli and Lanza, 2013) (Syer et al., 2013).

The diversity of mobile devices and platforms made native development of mobile applications challenging and costly, thus approaches like cross-platform development have emerged to reuse code across different mobile platforms by using various techniques including cross-compilation, virtual machines, and web technologies and platforms (Palmieri et al., 2012) (Heitkötter et al., 2013). Our case study, unlike the cross-platform development approach, was concerned with native development and sharing code with a desktop version of the application.

There are various types of software reuse possible (Ambler, 1998). Our study focused only on the reuse of source code in the form of a library or framework. However, the notions and concepts that we introduced in our case study for a multiplatform application development, such as PIP, PDP, and platform differences and variations (Cheon, 2019), are related with those of the established software engineering. For example, PIP and PDP are similar to a platform-independent model (PIM) and a platform-specific model (PSM), respectively, in model-driven software development (Brown, 2004). A PIM is a software model that is independent of the specific technological platform used to implement it, and is translated to a PSM, a model that is linked to a specific technological platform (Meservy and Fenstermacher, 2005). Software product line development has been widely adopted in professional software development to create a collection of similar software systems, known as a *product family*, from a shared set of software assets using a common means of production (Northrop, 2002). Several architectural styles are proposed for developing software product lines of Android applications (Durschmid et al., 2017). One core idea of the software product line development is identifying the commonalities and variabilities within a family of products (Coplien et al., 1998). In our case study, we used the commonality and variability analysis to identify platform differences and variations as well as the platform for the PIP.

6 CONCLUSION

We performed a small but realistic case study to measure quantitatively the degree of code reuse possible between Java and Android versions of an application. The case study application watches over the fluctuating prices of online products and thus helps a user to figure out the best time to purchase the products. To share code between the two platforms, we decomposed our application into two parts: the platform-independent part (PIP) and the platform-dependent part (PDP). The PIP is shared between the two platforms, and each platform has its own PDP to address platform-specific concerns. To determine code reuse achieved in our application, which is the main contribution of our work, we measured the size of our code with a simple metric counting the number of lines of code. Our finding is very promising in that we were able to achieve 40% and 37% code reuse for Java and Android versions, respectively, for a UI-intensive application. We also learned that the Android version requires 8% more code than the Java version. The degree of code reuse, of course, depends heavily on the types and degrees of platform differences and variations. We noticed several types of platform differences, each requiring a different technique to cope with it. It would be interesting future work to study the platform differences systematically to categorize them, to measure quantitatively their impacts on the code reuse, and suggest effective techniques to address them. As a side benefit, we also learned that multiplatform application development can improve the quality of an application dramatically. A platform-specific restriction or constraint may provide an opportunity to explore and test an application in a way that would be impossible or unnecessary on another platform, often exposing a potential issue or problem in the application. Platform differences and vari-

ations also encourage one to generalize APIs, especially those of the PIP, to accommodate them, thus making the code more reusable and extensible.

It is only fair that we should mention the overheads associated with multiplatform application development that we learned in our case study, such as configuration of tools, writing stub code, constant attention to subtle platform differences, and code size. In term of code size, our PIP/PDP multiplatform application requires 13% (Java) and 14% (Android) more lines of code than the single platform, monolithic versions.

REFERENCES

Abdalkareem, R., Shihab, E., and Rilling, J. (2017). On code reuse from StackOverflow: an exploratory study on Android apps. *Information and Software Technology*, 88:148 – 158.

Ambler, S. (1998). A realistic look at object-oriented reuse. *Software Development*, 6(1):30–38.

Apache Software Foundation (2019). Apache Maven project. https://maven.apache.org/, Last accessed on February 19, 2019.

Brown, A. W. (2004). Model driven architecture: Principles and practice. *Software and Systems Modeling*, 3(4):314–327.

Cheon, Y. (2019). Multiplatform application development for Android and Java. In *17th IEEE/ACIS International Conference on Software Engineering, Management and Applications, May 29-31, 2019, Honolulu, Hawaii*. To appear.

Cheon, Y., Romero, R., and Garcia, J. (2017). HifoCap: an android app for wearable health devices. In *Advances in Digital Technologies, Proceedings of the 8th International Conference on Applications of Digital Information and Web Technologies*, volume 295 of *Frontiers in Artificial Intelligence and Applications*, pages 178–192. IOS Press.

Coplien, J., Hoffman, D., and Weiss, D. (1998). Commonality and variability in software engineering. *IEEE Software*, 15(6):37–45.

Durschmid, T., Trapp, M., and Dollner, J. (2017). Towards architectural styles for Android app software product lines. In *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 58–62.

F. Bronsard, et al. (1991). Toward software plug-and-play. In *Proceedings of the 1997 Symposium on Software Reusability (SSR '97), Boston, MA*, pages 19–29.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley.

Gonzalez, H., Kadir, A. A., Stakhanova, N., Alzahrani, A. J., and Ghorbani, A. A. (2015). Exploring reverse engineering symptoms in Android apps. In *Proceedings of the Eighth European Workshop on System Security*, EuroSec '15, pages 7:1–7:7, New York, NY, USA. ACM.

Google LLC (2019). Firebase realtime database. https://firebase.google.com/products/realtime-database/, Last accessed on February 19, 2019.

Heitkötter, H., Hanschke, S., and Majchrzak, T. A. (2013). Evaluating cross-platform development approaches for mobile applications. In Cordeiro, J. and Krempels, K.-H., editors, *Web Information Systems and Technologies*, pages 120–138, Berlin, Heidelberg. Springer Berlin Heidelberg.

IDC (2019). Smartphone market share. https://www.idc.com/promo/smartphone-market-share/os, Last accessed on March 13, 2019.

Linares-Vásquez, M., Holtzhauer, A., Bernal-Cárdenas, C., and Poshyvanyk, D. (2014). Revisiting Android reuse studies in the context of code obfuscation and library usages. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 242–251, New York, NY, USA. ACM.

Martin, R. C. (2003). *Agile Software Development, Principles, Patterns, and Practices*. Prentice Hall.

Meservy, T. O. and Fenstermacher, K. D. (2005). Transforming software development: an MDA road map. *IEEE Computer*, 38(9):52–58.

Minelli, P. and Lanza, M. (2013). Software analytics for mobile applications-insights & lessons learned. In *European Conference on Software Maintenance and Reengineering, Genova, Itally*, pages 144–153. IEEE.

Mojica, I. J., Adams, B., Nagappan, M., Dienst, S., Berger, T., and Hassan, A. E. (2014). A large-scale empirical study on software reuse in mobile apps. *IEEE Software*, 31(2):78–86.

Northrop, L. M. (2002). SEI's software product line tenets. *IEEE Softw.*, 19(4):32–40.

Palmieri, M., Singh, I., and Cicchetti, A. (2012). Comparison of cross-platform mobile development tools. In *2012 16th International Conference on Intelligence in Next Generation Networks*, pages 179–186.

Ruiz, I. J. M., Nagappan, M., Adams, B., and Hassan, A. E. (2012). Understanding reuse in the Android market. In *2012 20th IEEE International Conference on Program Comprehension (ICPC)*, pages 113–122.

Sillars, D. (2015). *High Performance Android Apps: Improve Ratings with Speed, Optimizations, and Testing*. O'Reilly.

Speicher, T. and Cheon, Y. (2018). Composing a cross-platform development environment using Maven. In *Proceedings of the RCCS+SPIDTEC2 Workshop on Regional Consortium for Foundations, Research and Spread of Emerging Technologies in Computing Sciences, Juarez, Mexico*, pages 68–80. IOS Press.

SQLite Consortium (2019). Sqlite. https://www.sqlite.org/, Last accessed on February 19, 2019.

Syer, M. D., Nagappan, M., Hassan, A. E., and Adams, B. (2013). Revisiting prior empirical findings for mobile apps: An empirical case study on the 15 most popular open-source Android apps. In *Proceedings of the 2013 Conference of the Center for Advanced Studies on Collaborative Research*, CASCON '13, pages 283–297, Riverton, NJ, USA.

TIOBE (2019). TIOBE index for March 2019. https://www.tiobe.com/tiobe-index/, Last accessed on March 13, 2019.