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.

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 multiplatform development environment consisting of platform-specific tools. 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 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.

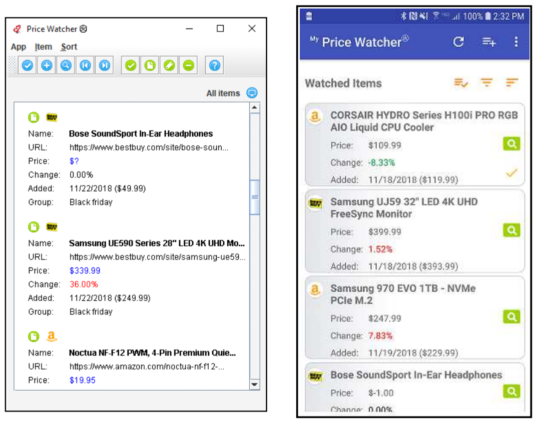
1 INTRODUCTION

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. The Android operating system is the dominating mobile platform of today, 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.

A *multiplatform application* is an application that is developed for and runs on multiple platforms. 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.

We answer our 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 natively 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 the average size (Minelli and Lanza, 2013), requiring a graphical user interface (GUI), data persistence, networking, and multithreading. 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. Our development environment propagates immediately changes made on the shared code using one IDE to other IDEs and platforms. Our design approach for code reuse 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

Cheon, Y., Chavez, C. and Castro, U.
Code Reuse between Java and Android Applications.
DOI: 10.5220/0007843702460253
In *Proceedings of the 14th International Conference on Software Technologies (ICSOFT 2019)*, pages 246-253
ISBN: 978-989-758-379-7



Name: Bose SoundSport In-Ear Headphones
URL: https://www.bestbuy.com/site/bose-soun...
Price: $?
Change: 0.00%
Added: 11/22/2018 ($49.99)
Group: Black friday

Name: Samsung UE590 Series 28" LED 4K UHD Mo...
URL: https://www.bestbuy.com/site/samsung-ue59...
Price: $339.99
Change: 36.00%
Added: 11/22/2018 ($249.99)
Group: Black friday

Name: Noctua NF-F12 PWM, 4-Pin Premium Quie...
URL: https://www.amazon.com/noctua-nf-f12-...
Price: $19.95

CORSAIR HYDRO Series H100i PRO RGB AIO Liquid CPU Cooler
Price: $109.99
Change: -8.33%
Added: 11/18/2018 ($119.99)

Samsung UJ59 32" LED 4K UHD FreeSync Monitor
Price: $399.99
Change: 1.52%
Added: 11/18/2018 ($393.99)

Samsung 970 EVO 1TB - NVMe PCIe M.2
Price: $247.99
Change: 7.83%
Added: 11/19/2018 ($229.99)

Bose SoundSport In-Ear Headphones
Price: $-1.00
Change: 0.00%

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

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.

The rest of this paper is structured as follows. In Section 2 we describe our case study application. In Section 3 we explain our development of the application, including the configuration of tools and design challenges along with our solutions. In Section 4 we assess code reuse by measuring it quantitatively and interpreting the measurements. In Section 5 we mention few related work, and we provide a concluding remark in Section 6.

2 CASE STUDY

The primary objective of our case study is to measure quantitatively the amount of code reuse possible between a Java application and its Android version. The secondary objective is to learn complications or issues associated with multiplatform application development—those related with not only design and coding but also configuration of tools for incremental development with continuous integration.

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.

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. Android SDK, for example, includes one particular open source JSON library while Java SDK does not. An open-source library or framework can be tightly integrated into the platform and be part of the platform's APIs. It is in a sense provided as a built-in feature of the platform, often modified significantly through the integration. An example is Android's support for SQLite, a lightweight, server-less relational database system. There are noticeable differences between 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 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.

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). 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 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

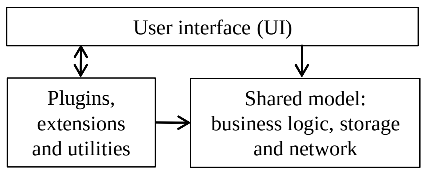
IDEA. For the development of both the Java application and the common code, any reasonable Java IDE including IntelliJ IDEA will work. However, since we are also interested in learning issues caused by diversity of tools, we opt for Eclipse.

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, 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.

Our primary design goal is to maximize code reuse between Java and Android versions. 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 such as platform-specific APIs (Cheon, 2019). The *platform-dependent part* (*PDP*) is the part of an application that does depend on a specific platform. 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.

We need 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, sharable code—has to accommodate the platform differences and variations. Figure 2 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.

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). 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 asking 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 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 3 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 An-


User interface (UI)
Plugins, extensions and utilities
Shared model: business logic, storage and network
Figure 2: Application architecture.

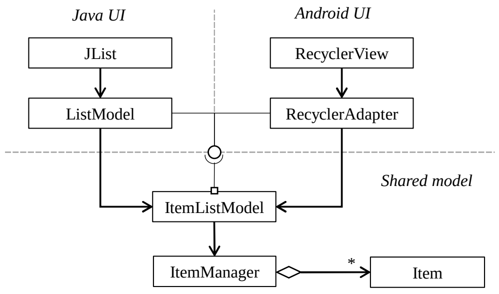


Figure 3: Plugging in platform-specific code.

droid. 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.

Another common 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. We use this approach to persist watched items in several different ways (see Section 4). 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.

4 ASSESSMENT

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 source code lines. We can group the PIP classes into four different groups for later analyses.

- Items: classes for the core business logic, consisting of Item, ItemManager and ItemListModel.
- Storage: classes to persist watched items. Three different approaches—file, local database (SQLite) and cloud storage (Google Firebase)—are used to study the impact of API differences on the code reuse.
- Network: classes for online stores, or websites, and for finding an item's price from its webpage.
- Others: utility and miscellaneous classes.

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

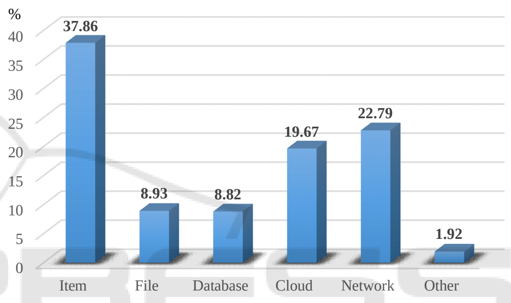


Figure 4: Distribution of PIP code.

It would be interesting to see how the code is distributed among these groups of classes (see Figure 4). 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 between the two platforms (see Section 3). 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 the complexities of the data persistence approaches themselves or their code reusability.

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

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. One side finding is that the Android PDP requires 14% ((3162 - 2779) / 2779 * 100) more code than the Java PDP. Android applications generally require more coding to address Android-specific concerns such as application lifecycles and screen orientation changes. The Android version of the application requires 8% more lines of code 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 concerns, 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 requires 3–6 times more code (see below for more discussion on this).

It would be very 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 5 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 6 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 5 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

Table 3: PDP code for interfacing with PIP.

Class	No. of Lines Java	No. of Lines 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

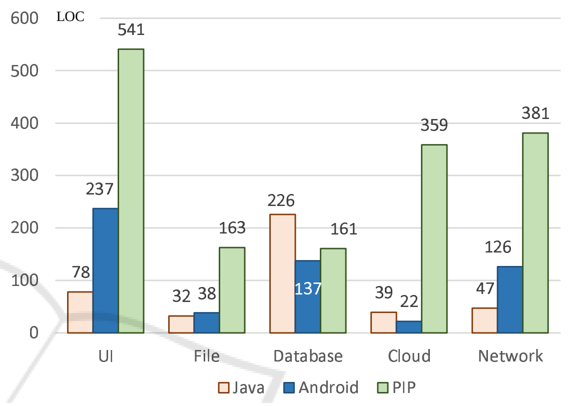


Figure 5: PIP features and their interfacing code.

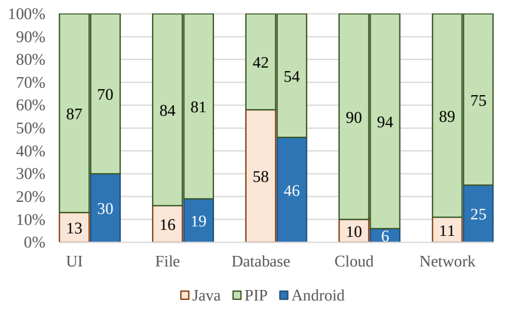


Figure 6: PIP and interfacing code in percentages.

Table 4: Complexity of data persistence in LOC.

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

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 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 6). 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 wrote our application incrementally by writing complete code in 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.

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	

4.3 Other Findings

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.

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 in 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.

An 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. The platform of 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.

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 used 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 case study to measure quantitatively the degree of code reuse possible between Java and Android versions of an application. For code sharing, 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, 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.

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.

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.

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 *4th International Conference on Mobile Software Engineering and Systems*, pages 58–62.

F. Bronsard, et al. (1991). Toward software plug-and-play. In *Symposium on Software Reusability, 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 *8th European Workshop on System Security*, pages 7:1–7.

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. Springer.

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 *11th Working Conference on Mining Software Repositories*, pages 242–251. 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 Software*, 19(4):32–40.

Palmieri, M., Singh, I., and Cicchetti, A. (2012). Comparison of cross-platform mobile development tools. In *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 *20th IEEE International Conference on Program Comprehension*, pages 113–122.

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 *Conference of the Center for Advanced Studies on Collaborative Research*, pages 283–297.