Multiplatform Application Development for Android and Java

Yoonsik Cheon

TR #19-42
April 2019

Keywords: multiplatform application, platform difference, software development, software engineering, 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

To appear in the *17th IEEE/ACIS International Conference on Software Engineering Research, Management and Applications*, Honolulu, Hawaii, May 29-31, 2019.

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

Multiplatform Application Development for Android and Java

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

***Abstract*—Software developers of today are under increasing pressure to support multiple platforms, in particular mobile platforms. However, developing a multiplatform application is difficult and challenging due to a variety of platform differences. We propose a native approach for developing a multiplatform application running on two similar but different platforms, Java and Android. We address practical software engineering concerns attributed to native multiplatform application development, from configuration of tools to software design and development process. Our approach allows one to share 37%~40% of application code between the two platforms as well as improving the quality of the application. We believe our approach can also be adapted to transforming existing Java applications to Android applications.**

Keywords—multiplatform application, platform difference, software development, software engineering, Android, Java

I. INTRODUCTION

Android and Java applications can be written in the same programming language. However, the similarity almost ends there, as there are significant differences between the two popular platforms of today in terms of application programming interfaces (APIs), software development kits (SDKs) and runtimes. A subtle platform difference can cause an application to malfunction or show a radically different behavior. Android applications are smaller than traditional applications with the average size of 5.6 kLOC [8], and the development of mobile applications tends to be driven by a single developer [14].

In this paper we propose an approach for developing an application that runs on both the Android platform and the Java platform. A *multiplatform application* is an application that is developed for and runs on multiple platforms. We use the term platform broadly to mean operating systems, runtimes, SDKs including APIs, and even design guidelines. We first identify and describe the software engineering challenges associated with multiplatform application development. The platform differences are of course the number one cause of all the challenges, and thus we identify various types of platform differences and variations between Android and Java. We then propose a development approach to address the challenges and identified platform differences.

Our proposed approach includes a development process, an overall application architecture and configuration of tools (see Section IV). Our process is iterative and incremental to better address the diversity of platforms as well as uncertainty of platform differences. Each build is incremental and a working build is delivered after each iteration. Our design approach is to separate platform-specific parts from the rest of the application. An application is decomposed into two distinct parts: a *platform-independent part* (*PIP*) and a *platform-dependent part* (*PDP*). The PIP is shared across platforms, and a separate PDP is written for each platform. Several techniques are suggested to separate the two parts cleanly and encapsulate platform differences in PDPs. We suggest to configure a development environment consisting of several platform-specific integrated development environments (IDEs) to support continuous integration [13].

We applied our approach to the development of a small but realistic application (see Section V). The finished application consists of 36~40 classes and 4604~4987 lines of manually-written Java source code. We were able to achieve 37%~40% code reuse in spite of the application being user interface-intensive. The case study confirmed that code reuse indeed depends heavily on the degree of API similarity, with a wide range of reuse percentages (52%~92%). We also learned that our approach supports the separation of concerns very well and provides valuable opportunities for improving the quality of an application. The configuration of tools was effective in supporting continuous integration and testing. It lets us to work on multiple projects simultaneously by simply switching between the platform-specific IDEs. In sum, our approach works well for a single developer and a small team of developers.

The rest of this paper is organized as follows. In the next two sections we identify and describe challenges associated with multiplatform application development, focusing on platform differences. In Section IV we explain our development approach, including a process, an application architecture, and tool support. In Section IV we evaluate our approach by performing a small case study. In Section VI we mention related work, and we conclude our paper in Section VII.

II. CHALLENGES

In this section we identity and describe briefly some of the software engineering challenges associated with multiplatform application development. We use the term platform broadly to mean a set of frameworks and tools to create a complete application.

- *Platform difference*. This is the fundamental reason why multiplatform application development is different from other development approaches. An application has to run on multiple platforms, and there are a variety of

platform differences and variations, including APIs, constraints, and even design guidelines (see Section III).

- *Design for difference*. The biggest design challenge is to accommodate platform differences and variations as well as maximizing code reuse across platforms. We need a software design model that supports variations as well as code reuse. Established software engineering principles and concepts like software components [2], software product lines [11], and design patterns [7] may be adapted and applied in the design of a multiplatform application. However, it is not clear exactly how they can be adapted and applied to various types of platform differences. A subtle platform difference may have a significant impact on the design of an application.

- *Configuration of tools*. It is crucial to have adequate tool support for any software development. Since multiple versions (variations) of an application are constructed—one for each platform—it is likely that a combination of platform-specific tools be used for multiplatform application development. The challenge here is to create sort of an integrated environment consisting of several platform-specific tools working in harmony. One key requirement for a development environment is to propagate changes made by one tool to the others immediately. Ideally, the environment should assist a programmer in detecting platform-related issues, identifying their causes, and propagating the fixes.

- *Development process*. We need to work on multiple projects, one for each platform and a library project. The library project is for developing the common code to be shared across platforms. We need a conceptual model to manage platform differences, work on multiple projects, and to incorporate variations as one of the key development elements.

III. Platform Differences

Android applications are written in Java; they can be written in other languages such as Kotlin and C/CC+, but Java is the most popular language at the time of writing. The APIs of Java and Android are similar for common libraries such as collections. However, the graphical user interface (GUI) frameworks of Java and Android are completely different, as Android offers its own framework for GUI programming. Android also introduces quite a few concepts and framework classes specifically for mobile applications. In this section we describe some of the noticeable differences between the two platforms from a programmer's perspective. We can categorize API differences in several ways, including:

- *Syntactic vs. semantic*. The difference can be purely syntactic, semantic, or both. We use the term a *syntactic interface* to mean the syntactic aspect of an API such as names and signatures, and a *semantic interface* to mean its behavior or meaning.

- *Built-in vs. third-party*. The difference can exist between APIs of the platform SDKs or third-party libraries and frameworks. There are several different ways to support third-party libraries and frameworks.

- *Restrictions and constraints*. A platform can restrict the way its APIs are used. These restrictions and constraints can be enforced by the platform either statically at compile time or dynamically at runtime.

- *Design guidelines*. Each platform has its own design guidelines—a set of principles and recommendations along with supporting APIs. The guidelines are most often about the design of UI but also pertain to the use of APIs and other features of the platform SDK.

An example of the syntactic/sematic difference is requesting the UI thread, called the *event dispatch thread* in Java, to perform an action or task. Java provides a static method named *SwingUtilities.invoakeLater(Runnable*) while Android defines a non-static method *Activity.runOnUiThread*(*Runnable*). These seemingly equivalent operations also have a subtle semantic difference. If the current thread is the UI thread, the requested action is executed immediately on Android; in Java, however, it is deferred until all pending events have been processed. This kind of subtle semantic differences often causes more trouble than a missing API, another common form of API differences.

Besides the APIs of the platform SDK, third-party libraries and frameworks are used heavily in application development, especially in Android applications [8]. Interestingly, there are also differences in the ways third-party libraries and frameworks are supported by platforms: (a) bundled in the platform SDK, (b) integrated with the platform SDK, and (c) provided as a separate platform-specific SDK. Examples include Android's support for JSON, SQLite databases, and Google Firebase cloud storage, respectively. Java SDK provides no direct support for JSON or SQLite, and Google offers a Java-specific Firebase SDK.

Perhaps, the two most noticeable constraints of the Android APIs are network operations and UI updates. Android disallows network operations on the main (UI) thread. The Android runtime throws a *NetworkOnMainThreadException* when an application attempts to make a network operation on its main thread. Android also prevents background threads from updating the UI. An application's main thread is solely responsible for updating the UI. There are no such restrictions enforced in Java.

An example of the design guidelines is string externalization. Android recommends one to externalize UI strings—to store them in XML files in special resource directories—so that it can pick an appropriate definition depending on the current language setting of the device. In fact, Android defines more than a dozen resource types, including color, image, layout, menu, string, style and animation, and provides a way to refer them. It is good practice to use the Android resource framework to separate the localized aspects of an application from the core functionality coded in Java. There is no such, or similar, resource framework offered by the Java SDK.

IV. Our Approach

In this section we propose to develop a multiplatform application incrementally and iteratively by continuously integrating and testing its code written on different platforms. We describe our proposed approach, including a process, an application architecture, and tool support.

A. Process

A subtle difference of platforms can have a big impact on the design of a multiplatform application. Due to the diverse nature of platform differences and variations—not only their types but also their complexities and delicacies—it is hard, or sometimes impossible, to identify and know all the platform differences at early stages of the development. We therefore propose an iterative and incremental approach in which each software build is incremental in terms of features and a working build is delivered after each iteration (see Fig. 1). The final build of course implement all the required features.

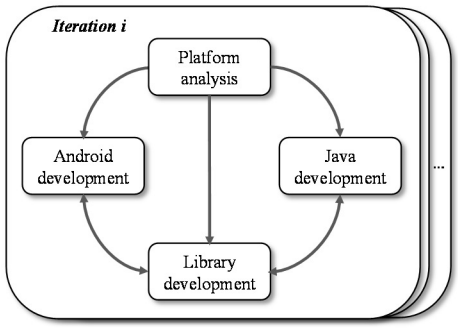

Fig. 1. Iterative development

Each iteration consists of four different activities. Each iteration starts with a platform analysis activity to identify differences and variations of the platform APIs for the feature under development in that iteration. This activity provides a basis for the other three activities, all of which are concerned with design and coding. In each iteration we need to work on three different projects: an Android project, a Java project, and a library project. The library project is for developing the common code to be shared between platform-specific versions, or variations, of the application. Our approach allows different sequences in which the three projects can be performed, including of course simultaneous work. The development environment to be configured later in this section allows us to work on all three projects at the same time by switching between platform-specific IDEs. However, we found that two sequences work best: (a) library and then platforms and (b) platform, library, and then platform. The *library first* approach works well for familiar features with little platform difference, and the *platform-library-platform* for other cases. The underlying idea of the second approach is to first develop an increment for one platform, derive reusable code (library) from it, and then apply the library to the development of the other platform. The strength of this approach is that it lets us to tackle a concrete and specific problem first and then generalize the solution to solve similar problems. Designing reusable classes is more difficult and time consuming than designing classes for one specific problem [1].

B. Application Architecture

The architecture and design of a multiplatform application should accommodate platforms differences and variations as well as maximizing code reuse across platforms. A natural architecture therefore is to separate platform-specific parts from the rest of the application. We suggest to decompose an application into two distinct parts: a platform-independent part and a platform-dependent part. The *platform-independent part* (*PIP*) is the part of an application that doesn't depend on specifics of a particular platform. In the model-view-controller (MVC) architecture, the model is a good 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. The PIP code is written once and shared across platforms whereas different PDP code is written for each platform.

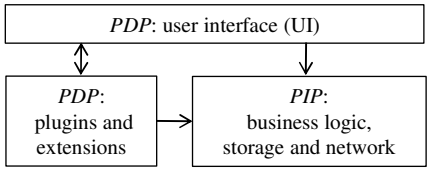

Fig. 2. Architecture for multiplatform applications

Fig. 2 shows our suggested architecture for a multiplatform application. The most noticeable PDP is the UI of an application. Remember that Android provides its own GUI framework along with Android-specific concepts and framework classes. The other PDP encapsulates platform differences and variations of the application for the PIP. The PIP is the functional core, or business logic, of the application and may also include storage and network. As expected, the PIP doesn't depend on PDPs. We suggest several techniques to accommodate platform differences and variations in the design. The guiding principle is to have clearly defined interfaces and employ loose coupling between the two parts.

- *Required interface*. The PIP needs to interact with the PDP. We let the PIP depends on an abstraction of the PDP, not its concrete implementation, by applying the *dependency inversion principle* [6]. The assumption that the PIP makes about the PDP is coded explicitly in the form of a required interface [2]. In a sense, we plug in platform-specific code—a class that implements the required interface—to the platform-neutral framework.
- *Inheritance and hook*. The PIP provides an abstract class with hook methods to be overridden by a subclass in the PDP. That is, the PIP defines a skeleton algorithm and lets the PDP fill out the details in a platform-specific way. This can be coded by applying several well-known design patterns such as template method, strategy, factory method [7].
- *Parameterization*. A platform difference or variation can be parameterized. The PIP, instead of accessing a PDP feature directly, receives it as a parameter. The parameters can be simple values, structured data and objects, and behaviors (lambdas). The required interface can be viewed as a special form of parameterization.
- *Interface cloning*. This is a simple technique to get rid of a platform difference. If an API is provided only in one platform, we can clone it in the other platform—

define a class of the same syntactic interface but coded in a platform-specific way. The API now become a common feature of both platforms and thus can be used in the implementation of the PIP.

- *Interface unification*. The APIs of the two platforms can be merged and unified to get rid of their differences. This is a generalization of the interface cloning above.

C. Tool Support

It is crucial to have adequate tool support for multiplatform application development. Since two versions of an application are developed, the development environment logically consists of three IDEs: two platform-specific IDEs and one for developing the PIP. Thus, it is very likely that a combination of several tools be used for multiplatform application development. We configure a custom multiplatform application development environment by composing platform-specific IDEs and tools as suggested in [13].

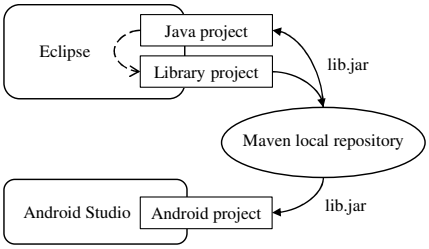

Fig. 3. Example development environment

Fig. 3 shows one possible configuration of a development environment consisting of Eclipse and Android Studio. One key requirement for incremental and iterative development of a multiplatform application is to propagate changes immediately from one IDE to others. We can use Apache Maven, a software project management and build tool, to share code and propagate changes in the form of a library [13]. 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 immediately to the Android project. The Gradle build tool of Android Studio understands Maven repositories. For the Java project we can also make it reference, or depend on, the library project.

V. Evaluation

We performed a small but realistic case study to evaluate our approach both quantitatively and qualitatively. We developed an application to watch over the fluctuating prices of online products and thus help a user to figure out the best time to purchase them (see Fig. 4). We developed the application iteratively in several increments: single item, multiple items, data persistence, and network. We used the library-first approach for the first increment and the platform-first approach for the others. Our development environment consisted of Eclipse and Android Studio as shown in Fig. 3 (see Section IV.C), and the library project was developed in Eclipse. The environment was effective for supporting continuous integration and testing, especially the integration of the PIP to the two PDPs. We oftentimes worked on all three projects simultaneously—e.g., to refine the PIP and test it on both platforms—by switching between Eclipse and Android Studio.

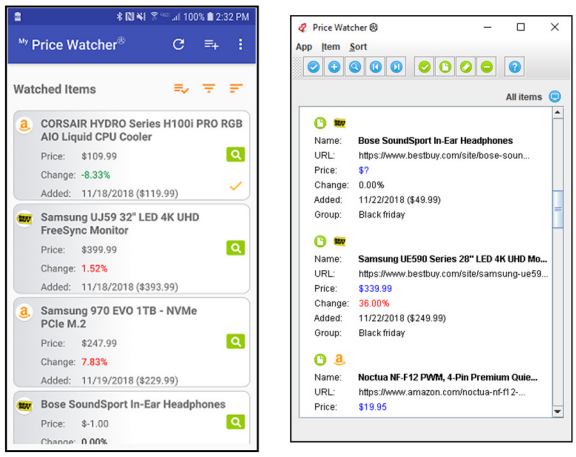

Fig. 4. Screenshots of Price Watcher (Java and Android)

The complete Android application consists of 40 classes and 4987 lines of source code (LOC), and the Java versions consists of 36 classes and 4604 LOC. The PIP accounts for 37% and 40% of the Android and the Java application code, respectively (see Table 1). That is, we achieved 37% and 40% code reuse for Android and Java, respectively. Android version has 8% more code than Java.

Table 1. Sizes of application code

App	Part	No. of Classes	No. of Lines	*Percent (%)*
Android	PDP	24	3162	63
	PIP	16	1825	37
Java	PIP			40
	PDP	20	2779	60

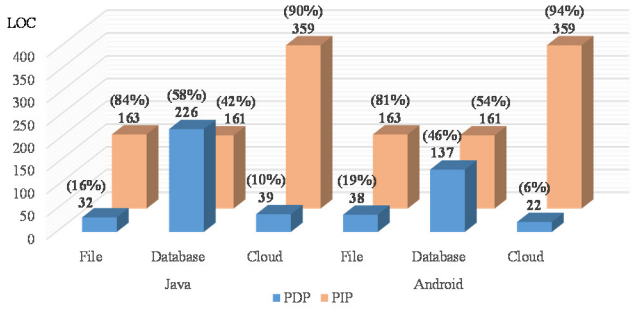

Fig. 5. Code sizes of different data persistence schemes

How do the API differences affect code reuse? To study this, we implemented three different data persistence schemes: file, database (SQLite) and cloud storage (Google Firebase).

Fig. 5 shows the amount of code written for each of these data persistence schemes. We were able to achieve a high degree of code reuse for the file (83%) and the cloud storage (92%). For these two schemes, only small amount of code was written on each platform (PDP), because their Android and Java APIs are very similar. For the database, however, more code (52%) was written in PDPs. As said previously, Android offers SQLite databases integrated with its framework, and thus its API is significantly different from the JDBC-based SQLite of Java.

The PIP/PDP architecture slightly increased the size of the application code. Monolithic versions—code written without the PIP/PDP separation—have less classes and lines of code, and the size overheads of the PIP/PDP separation are 14% and 13% for Android and Java, respectively.

Our development approach provided valuable opportunities for improving the quality of an application. Android platform features such as screen orientation changes allowed us to explore and test our application in a way that would be impossible or unnecessary for a Java application, often exposing potential issues or problems in our application. We worked on, or reviewed, the same or derived code (PIP and PDPs) several times, each with a different perspective—either as a service provider or a consumer. Multiplatform development encouraged us to generalize our APIs, especially those of the PIP, to address and accommodate platform differences and variations. In fact, even platform restrictions and constraints contributed positively to the creation of a more reusable and extensible application. However, one downside of our PIP/PDP separation is that the PIP is written in the common denominators, or shared traits, of the both platforms. That is, the PIP platform is the intersection of the two platforms, Java and Android.

VI. Related Work

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 such as cross-compilation, virtual machines, and web technologies [5] [12]. Unlike these approaches, our work is concerned with native development of a mobile application and sharing code with its desktop version written in the same programming language, where the development processes and practices can be quite different [8] [14]. It is said that the practice of software reuse is high among mobile application developers [9]. One study even reported that 61% of Android application classes appeared in two or more other applications [10]. However, we found no publish work measuring code reuse between Android and Java applications.

Our notions of PIP and PDP are similar to a platform-independent model and a platform-specific model, respectively, of model-driven software development [3]. Our approach is also related with the software product line development that aims to create a collection of similar software systems, called a *product family*, from a shared set of software assets using a common means of production [11]. The key is to identify the commonalities and variabilities within a family of products [4]. In our approach, the PIP is a shared asset, and the platform APIs are the variabilities.

VII. Conclusion

We proposed an approach for developing a multiplatform application for Java and Android. We showed that the two seemingly equivalent platforms of Java and Android have significant API differences that pose challenges in application development. Our solution to the challenges is to develop an application incrementally and iteratively by including an analysis of platform differences as a key component of the development process. We showed the effectiveness of our approach by applying it to a small but realistic case study. Our approach not only allowed us to achieve 37%~40% code reuse but also provided valuable opportunities for improving the quality of the application. It was also shown that code reuse across platforms depends heavily on the similarity of platform APIs. The main contributions of our work include (a) identification of various platform differences between Java and Android, (b) design techniques for accommodating platform differences, (c) notions of a platform independent part (PIP) and a platform dependent part (PDP), (d) an application architecture based on the PIP/PDP, and (e) configuration of a development environment consisting of a set of platform-specific tools.

References

[1] R. E. Johnson and B. Foote, "Designing reusable classes," Journal of Object-Oriented Programming, vol. 1, no. 2, pp. 22-35, June/July 1988.

[2] F. Bronsard, et al., "Toward software plug-and-play," Symposium on Software Reusability (SSR '97), Boston, MA, May 1997, pp. 19-29.

[3] A. W. Brown, "Model driven architecture: Principles and practice," Software and Systems Modeling, vol. 3, no. 4, pp. 314–327, Dec. 2004.

[4] J. Coplien, D. Hoffman and D. Weiss, "Commonality and variability in software engineering," IEEE Software, vol. 15, no. 6, pp. 37–45, 1998.

[5] H. Heitkotter, S. Hanschke and T. A. Majchrzak, "Evaluating cross-platform development approaches for mobile applications," Web Information Systems and Technologies, pp. 120–138, Springer, 2013.

[6] R. C. Martin, Agile Software Development, Principles, Patterns, and Practices, Prentice Hall, 2003.

[7] E. Gamma, R. Helm, R. Johnson and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison Wesley, 1994.

[8] 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.

[9] I. J. Mojica, B. Adams, M. Nagappan, S. Dienst, T. Berger and A.E. Hassan, "A large-scale empirical study on software reuse in mobile apps," IEEE Software, vol. 31, no. 2, pp. 78–86, 2014.

[10] I. J. Mojica, M. Nagappan, B Adams and A. E. Hassan, "Understanding reuse in the Android market," IEEE International Conference on Program Comprehension (ICPC), pp. 113–122, 2012.

[11] L. M. Northrop, "SEI's software product line tenets," IEEE Software, vol. 19, no. 4, pp. 32–40, 2002.

[12] M. Palmieri, I. Singh and A. Cicchetti, "Comparison of cross-platform mobile development tools," International Conference on Intelligence in Next Generation Networks, pp. 179–186, 2016.

[13] T. Speicher and Y. Cheon, "Composing a cross-platform development environment using Maven," Workshop on Regional Consortium for Foundations, Research and Spread of Emerging Technologies in Computing Sciences, Juarez, Mexico, Nov. 8-9, 2018, pp. 68-80.

[14] M. D. Syer, M. Nagappan, A. E. Hassan and B. Adams, "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, CASCON '13, pp. 283–297, Riverton, NJ, 2013.