9-2020

Creating Flutter Apps from Native Android Apps

Yoonsik Cheon

Carlos Chavez

Comments:
Technical Report: UTEP-CS-20-95

Creating Flutter Apps from Native Android Apps

Yoonsik Cheon and Carlos Chavez

TR #20-95
September 2020

Keywords: cross-platform application, mobile app; Android, Flutter, Java

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

Submitted for publication.

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

Creating Flutter Apps from Native Android Apps

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

Carlos Chavez
Department of Computer Science
The University of Texas at El Paso
El Paso, Texas, U.S.A.
cvchavez92@gmail.com

ABSTRACT
Flutter is a development framework for building applications for mobile, web, and desktop platforms from a single codebase. Since its first official release by Google in less than a couple of years ago, it is gaining so much popularity among mobile application developers, even being regarded as a game-changer. There are, however, millions of existing native apps in use that meet the requirements of a particular operating system by using its SDK. Thus, one natural question to ask is about rewriting an existing native app in Flutter. In this paper, we look at the technical side of the above question by considering Android apps written in Java. We create a Flutter version of our existing Android app written in Java to support both Android and iOS by rewriting the entire app in Flutter. We share our development experience by discussing technical issues, problems, and challenges associated with such rewriting effort. We describe our approach as well as the lessons that we learned.

CCS Concepts
• Software and its engineering → Software creation and its management • Software and its engineering → Software notations and tools.

Keywords
cross-platform application; mobile app; Android; Flutter; Java.

1. INTRODUCTION
A mobile app development today generally implies the development of two versions, one for Android and the other for iOS. An alternative approach is to create a so-called *cross-platform app* that runs on multiple platforms. Flutter is a cross-platform development framework for building apps that runs on multiple platforms including both Android and iOS [7]. Since its official release by Google in December 2018, it is gaining so much popularity among mobile app developers. It is even being regarded as a game-changer by some developers. One of the reasons for its rapid popularity is that it is claimed that Flutter has solved the performance issues associated with existing cross-platform development approaches such as Apache Cordova, Microsoft Xamarin, and Facebook's React Native [2].

There are about three million Android apps available today through the Google Play Store. It is said that Android apps are smaller than traditional applications with the average size of 5.6K source lines of code (SLOC) [12], and the development of mobile apps tend to be driven by a single developer [17]. Flutter could be a good solution for an individual app developer or a small team of developers to support multiple platforms, since you write and maintain a single code base for multiple platforms. Considering the enormous number of apps already in use today, it is natural to ask whether it is practical to rewrite an existing native app in Flutter. One challenging aspect may be platform differences. It is known that a subtle platform difference can cause an application to malfunction or show radically different behavior [1].

In this paper, we consider the question of rewriting an Android app in Flutter to support both Android and iOS. There are a lot of questions to be answered and details to be worked out, including the following. Is it possible and practical to systematically rewrite, translate, or derive a Flutter app from a native Android app written in Java? What are the challenges, issues, and problems associated with such development efforts? Are there any sorts of reuse possible? What would be a good development approach or process? One good way to study this kind of questions is to consider and solve a concrete problem. We create a Flutter app from an existing native Android app written in Java. We rewrite the entire app in Flutter.

A subtle platform difference can have a great impact on the design and coding of a cross-platform app. One challenge of rewriting an Android app in Flutter, therefore, is to know and address the platform differences. There are different types of platform differences between Android and Flutter, including programming languages, libraries, application programming interfaces (APIs), platform constraints, and even design guidelines. We identified major differences between Android and Flutter platforms that are likely to require design and coding considerations, such as static vs. optional typing, multithreading vs. isolates (concurrency), imperative vs. reactive programming styles, and declarative vs. imperative user interface definitions. As it is almost impossible to know all the platform differences in advance, we use an iterative and incremental approach for our development. We hope this development approach allows us to address platform differences with minimal impact as they show up during iterations.

The design of the Flutter app was a reflection of the Android app's design, a model-view-control (MVC) design, and most coding efforts were spent on the view part. This is because unlike Android, a Flutter user interface is coded, not declared in a markup language such as XML, as a composition of (user-defined) widgets. Besides, the user interface is reactive in that it reacts to, or reflects, the state changes. Rewriting the model classes in Dart, the language of Flutter, was rather straightforward due to the similarity of collections APIs between Dart and Java. It was essentially a manual translation of constructs of one language to equivalent ones of another language. The final Flutter app is about 37% smaller than the Android app in SLOC -- 4351 lines of

SAC'21, March 22 –March 26, 2021, Gwangju, South Korea

DOI: xx.xxxx/xxx_x

Dart code vs. 6949 lines of Java code. We believe this small code size is due to the Flutter framework itself as well as the Dart language; both provide higher levels of abstraction. Interestingly, the percentages of the user interface code over the whole code are similar between the two apps with 73% and 74% for Android and Flutter, respectively. Android user interfaces are declared in XML, so this means that a lot more control code needs to be written on Android. Ironically, it may mean more efforts on the user interface part on Android -- designing a user interface in XML and writing its control code.

So, will it be a good idea to start converting Android apps to Flutter? Besides the smaller code size mentioned above, there are definitely benefits of rewriting an Android app in Flutter, mostly due to the cross-platform nature of the Flutter framework, such as productivity, shorter development time, cost-effectiveness, and larger user base [15]. However, we think one also needs to consider the maturity of the Flutter platform, its support tools, and the community. For example, there is no built-in, official, or standard way of externalizing app resources such as images and user interface strings. There is no platform-level support for handling the diversity of devices and user bases, e.g. picking up an image automatically based on the device's screen quality or user interface strings based on the current language setting of the device. There is no support for so-called *build variants* to expose an app in different versions, or flavors, from a single project. In a reactive programming model of Flutter, you need to manage the state that may change due to user interactions or background tasks. There are numerous approaches proposed, but there is no one officially supported by Flutter SDK for implementing sort of a global state, a state shared by multiple (unrelated) widgets. Similarly, there are a large number of third-party libraries, called *packages*, to access the underlying platform's features in a platform-neutral way. In most cases, however, there is a multitude of available packages contributed by different developers. One may become successful and be part of the Flutter SDK or de facto standard, but you don't know which one as you have to pick one for your app. It should be also noted that some packages require you to perform platform-specific settings or configurations, which may require some level of knowledge on both Android SDK and iOS SDK. Therefore, it is very important to study carefully whether the Flutter platform and its tools support stably all the features needed by your apps.

The rest of this paper is organized as follows. In Section **Error! Reference source not found.** below we give a quick overview of Flutter and its language Dart to provide background knowledge for reading this paper. We assume the readers are familiar with Android app development. In Section **Error! Reference source not found.** we describe the Android app to be rewritten in Flutter -- a quiz app being used in classrooms. In Section **Error! Reference source not found.** we explain the development of a Flutter version of the app, starting with the challenges, continuing to the approach, and concluding with the design and implementation. In Section **Error! Reference source not found.** we share our findings and the lessons learned.

2. BACKGROUND

Flutter is Google's user interface (UI) toolkit for building natively compiled applications for multiple platforms from a single code base. With Flutter, you can write a single code base that works on Android and iOS equally well. Unlike a native app that meets the requirements of a particular operating system and its SDK such as Android or iOS, a Flutter app is compatible with multiple operating systems and can run on both Android and iOS smartphones and tablets. The key idea of Flutter is to render its own UI components called *widgets* that are natively compiled. That is, there is no use of native UI components or asking the OS to render them. This also has several side benefits. You can add new widgets easily without worrying about the support of the underlying platforms. It is also possible to introduce design-specific widgets. In fact, Flutter provides two sets of widgets: Material design widgets of Android and Cupertino design widgets of iOS. The Flutter framework consists of four main components: (a) Dart platform, the language of Flutter, (b) Flutter engine responsible for rendering the UI, (c) foundation library providing interfaces over the native SDKs, e.g., to launch a camera app consistently across platforms, and (d) widgets, UI components written in Dart. The Flutter engine is C++ code based on the Skia graphics engine, an open-source 2D graphics library (see https://skia.org/), and is compiled into the app itself.

Flutter apps are written in the Dart language [3], a modern object-oriented programming language created by Google. The language was officially released in November 2013, but it experienced the most growth in recent years mainly driven by Flutter. Dart provides several interesting features including:

- Pure object-oriented language in C-style syntax with automatic garbage collection. In Dart, everything is an object. Even a null is an object. Therefore, all data are treated uniformly, e.g., no need for autoboxing and unboxing of primitive values.
- Class-based, or classical, language with features like abstract classes, interfaces, mixins, and reified generics. Every object is an instance of a class, and every class implicitly defines an interface that describes what methods are available on its interface. There is no explicit interface declaration syntax.
- Flexible type system called *optional typing*. Dart provides a productive balance between static and dynamic typing in that types are syntactically optional and do not affect runtime semantics. One can choose the level of type checking from static to dynamic.
- Isolates for concurrency. Instead of the popular thread model of Java and other languages, Dart introduced a message-based model for concurrent programming.
- Compiled to native code (ARM or x86), Dart virtual machine, or JavaScript.

We will discuss some of the Flutter and Dart features in the following sections.

3. MOQUIZ ANDROID APP

MoQuiz is a mobile app for classroom use, allowing students to take quizzes created by the instructor. The app was originally started as a small class project in the courses taught by the first author of this paper years ago, and it is now being used in all his courses. It evolved over the years to work with newer versions of Android operating systems as well as to provide additional features such as checking course grades and accessing lecture notes. Figure 1 shows the main screens of the MoQuiz app -- screens for taking a quiz and reviewing the result.

The MoQuiz application has a client-server architecture (see Figure 2). The server performs all the functional process logic, or business rules, of the application. It authenticates users, provides quizzes, and grades them as well as storing the results along with

grading review requests if any. The client is lightweight in that its main role is to provide a graphical user interface to the user. Initially, the Android app was the only client and thus it also implemented some of the business rules. Later, a Web client was added for those users who don't have Android devices, and most of the business rules were pushed back to the server. Besides the usual advantages of a modular software system with well-defined interfaces, the architecture allows any of the components to be upgraded or replaced independently in response to changes in requirements or technology. In fact, we were able to evolve the Android app to adapt to newer versions of the operating system with no or minimal impact on the other modules. It also turned out that the architecture made the development of a Flutter version of the mobile app easier, which is the main focus of this paper.

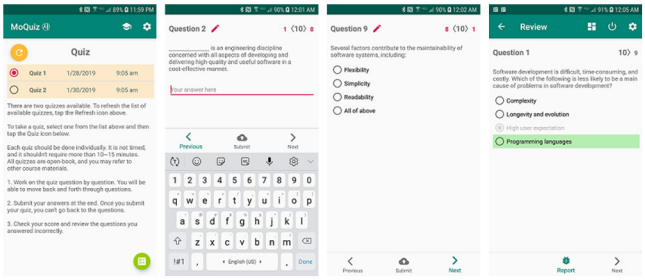

Figure 1. Sample screens of the MoQuiz Android app

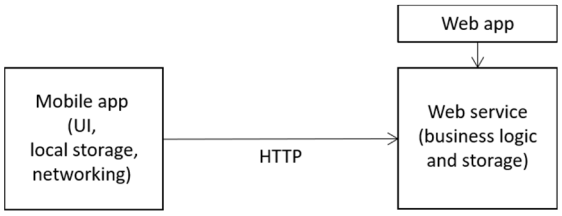

Figure 2. Architecture of the MoQuiz application

Table 1. Code size of the MoQuiz Android app

Components		**No. of Classes**	*Percent (%)*	**No. of Lines**	*Percent (%)*
Model		14	*30.43*	1343	19.33
Network		3	6.52	519	7.47
UI	General	11	23.91	2540	36.55
	Question	4	8.70	421	6.06
	Setting	7	15.22	535	7.70
	Grade	5	10.87	973	14.00
	Lecture	2	4.35	618	8.89
	All	29	63.04	5087	73.02
Total		46	100.00	6949	100.00

The implementation of the Android app consists of 48 Java classes with 7398 source lines of code (SLOC) including program comments (see Table 1). Most of the code (73%) is concerned with the user interface, and this is expected because the business logic of the app is implemented and provided by the server as a Web service. The Android app also scrapes lecture notes and course grades from the course website, but these functionalities are not pushed back to the server for some non-technical reasons. The source code includes two versions, or *build variants* in the Android terminology, of the app, one for the students and the other for the instructor. The instructor version, among other features, can show the course grades of all students along with statistical data in graphical forms.

4. MOQUIZ FLUTTER APP

4.1 Challenges

Creating a Flutter app from a native Android app can be challenging due to several factors such as platform differences and the cross-platform nature of Flutter. There are many differences between the two platforms, including programming languages, application programming interfaces (APIs), platform constraints, and design guidelines. Both Java and Android are classical object-oriented languages in that every object is an instance of a class, however, there are many differences between the two in terms of language features and constructs. Some of the differences make it easy to translate Java code to Dart. For example, Dart's reified generics -- objects of a generic type carry their type arguments with them at runtime -- make it straightforward to create instances of type arguments, which is a lot more involved in Java. Below we highlight some of the language and platform differences between Android and Flutter.

Static vs. optional typing. As mentioned in Section **Error! Reference source not found.**, Dart supports a flexible type system called *optional typing*. As the name indicates, a type declaration is optional and isn't required, but it really means that a programmer can pick the level of type checking from static to dynamic. The system uses a combination of static type checking and runtime checks to ensure that a variable's value matches the variable's declared type. Since Java is a statically typed language, one needs to decide the level of type checking to be done on the translated Dart code.

Imperative vs. reactive programming style. Flutter's user interface is *reactive* in that a widget is automatically rebuilt and redrawn when the state changes. It is radically different from the imperative style of Android in which a programmer creates a widget and sets its properties explicitly. In a sense, Flutter lets you define your user interface as a function of the current state. When the state changes, a new user interface -- which is a widget in Flutter -- is created and drawn. There is no sequence of mutator (or set) commands to configure or update a widget. You only need to manage the state that may change due to user interactions or background tasks.

Declarative vs. imperative definition of user interfaces. An Android user interface is usually defined declaratively in XML files, called *layout* resource files. For this, Android provides an XML vocabulary that corresponds to the view and widget classes. In fact, you use a tool to build a user interface layout by dragging widgets and other user interface elements into a visual design editor instead of writing the layout XML by hand. Unlike other mobile platforms, Flutter is unique in the way the user interface is expressed. You use the same Dart language to express an app's user interface as well as its behavior. That is, there is no separate markup language to express the user interface. The user interface is coded imperatively, and this user interface-as-code approach often means deeply nested code, i.e., deeply nested widget trees.

Cross-platform nature of Flutter. Android native apps are developed specifically for the Android operating system, and they may use features that are available only on Android or may rely on implicit assumptions that are valid only on Android. Flutter apps, however, are cross-platform and should run on both Android and iOS. A good example of an implicit assumption is the system back button. An Android device has a back button, either software or physical (see Section 7.2.3 of [1]). The back button is used to navigate, in reverse chronological order, through the history of screens that the user has recently worked with. It is common to design an Android user interface by relying on the existence of this back button, as it is also used to dismiss dialogs, popups, and the onscreen keyboard. Unlike Android, however, an iPhone doesn't provide a dedicated back button.

4.2 Approach

A subtle platform difference can have a great impact on the design and coding of a cross-platform app. This is also true when the app is created based on, or translated from, an existing native app. There are numerous platform differences of different types, complexities, and delicacies, and it is difficult to know all the platform differences in advance or at the early stages of the development [4]. We, therefore, use 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 Figure 3). This will allow us to address platform differences with minimal impact as they show up during iterations.

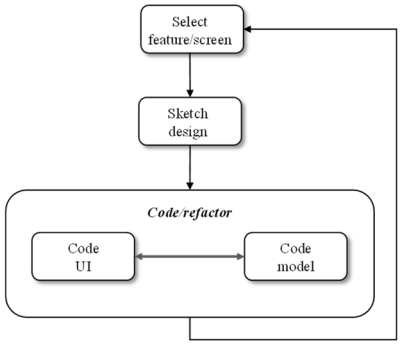

Figure 3. Iterative approach

We start each iteration by identifying a feature to implement. Since the app is essentially a client user interface with most of the business logic implemented and provided by the server in a form of Web service, a feature is closely related to one or more screens. As expected, we first worked on iterations for core features, or screens, such as showing questions and reviewing grades, and then moved to features like login, settings, alerts, and notifications. Lastly, we addressed add-on features such as course grades and lecture notes that require Web scraping. Before coding, we sometimes sketched a bit of design or made detailed design decisions. However, since we used the design of the Android app for both the user interface and other code, we were able to jump directly to coding in most iterations. The user interface and the model were coded in synchrony, but most coding efforts were spent on the user interface.

4.3 Design and Implementation

As mentioned above, we adapted the design of the Android app and spent most of our detailed design and coding efforts on the user interface part. As an example, consider the class diagram shown in Figure 4. It shows the main classes involved in the implementation of the questions screen that displays all the questions of a quiz and prompts the user for answers sequentially, one question at a time (see sample screenshots in Figure 5). The structure of model classes shown in shaded boxes on the right side is identical to those of the Android app, as the data to be shown and the way they are obtained from the server is the same regardless of the platforms. A quiz consists of a collection of questions, and there are different types of questions such as fill-in-the-blank questions and multiple-choice questions. The WebClient class is for obtaining quizzes from the server as well as scraping Web documents from the course website.

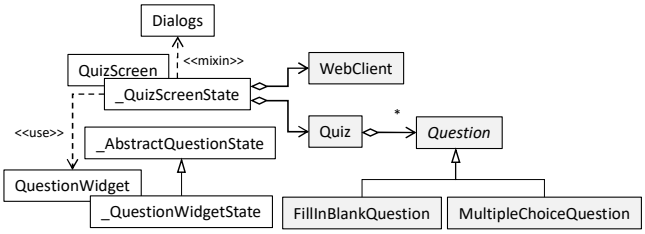

Figure 4. Design of the questions screen

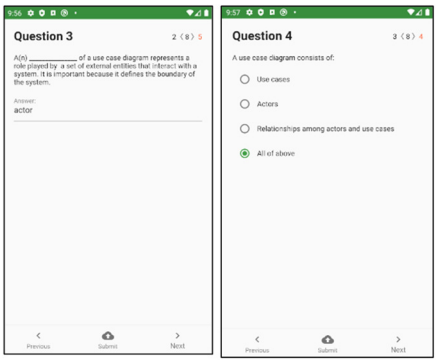

Figure 5. Screenshots of the questions screen

The user interface classes, or *widgets* in the Flutter terminology, are shown on the left side of the diagram. The QuizScreen class is the main widget class responsible for the whole screen to show all the questions of a quiz using a bottom navigation bar (see Figure 5). It is a stateful widget to show an appropriate question of the quiz when the user clicks the previous or the next button of the bottom navigation bar. It also submits the quiz to the server for grading when the user clicks the submit button in the navigation bar. For this, it uses the functionality provided by the WebClient class. In Flutter, a widget is either stateful or stateless. A stateful widget is dynamic in that its appearance can be changed when the user interacts with it. The state of a stateful widget is stored in a separate state object. The private class _QuizScreenState is the state class of the QuizScreen widget class; in Dart, if a name starts with an underscore, it is private to its library (file). It uses the QuestionWidget widget class to actually show a question. The QuestionWidget class is also a stateful widget because the user can select an option of a multiple-choice question or fills the blank of a fill-in-the-blank question and thus its state can change. The state is of course stored in a separate state class named

_QuestionWidgetState. This state class is a subclass of an abstract state class named _AbstractQuestionState. This little class hierarchy was introduced because there is another widget class similar to the question widget class to display not only a question and the user response but also the correct answer. Besides the QuestionWidget class, the quiz screen widget also depends on the Dialogs class, a utility mixin providing several forms of dialogs such as confirmation, alert, and progress. A *mixin* is a Dart program module similar to a class but allows to reuse code across unrelated classes.

It would be instructive to show the corresponding design of the Android app (see Figure 6). As mentioned above, we have the same structure for the model classes; there are also similarities in detailed designs – data structures and algorithms – as well. As expected, the designs of the user interface part are rather different for several reasons. In Android, the user interface layout – composition of views and widgets – are defined in an XML file, thus the user interface classes shown in the diagram are actually control code whose primary roles are finding views and widgets, setting their properties, and registering event handlers for them. Since Flutter adopted the user interface-as-code approach, all the user interface classes shown in the diagram are indeed (user-defined) widget classes, and the diagram shows the composition of the widgets. You always define widgets in Flutter, which is unusual in Android. In summary, the Flutter user interface classes do what the Android user interface-related classes plus the XML layout files do.

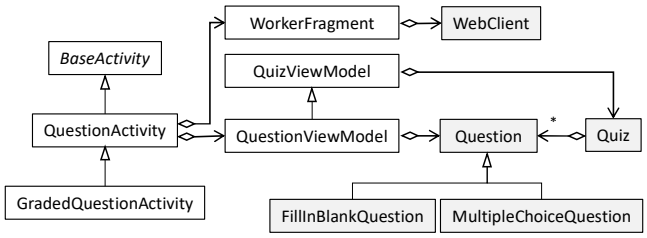

Figure 6. Android version of the design

Another noticeable difference between the two designs is the use of framework-specific programming concepts or styles. In Figure 6, you can see a group of classes that work as middlemen, or data binders, between the view and the model. These so-called *view model* classes store and manage user interface-related data in an activity lifecycle conscious way; an *activity* is a unit of Android apps responsible for a single screen. What this means is that the data stored in these classes survive configuration changes such as screen rotations, and thus they are immediately available to the new user interfaces created due to configuration changes, e.g., the landscape or portrait mode user interface after a screen orientation change. On Android, when there is a configuration change such as a screen orientation change, the current activity is paused, stopped, and destroyed and a new instance is created possibly with a different configuration, e.g., a landscape layout. The WorkerFragment class is a utility class playing a similar role as the view model classes but for computation. It preserves a long computation or an uncertain operation such as a network operation when there is a configuration change. In Flutter, we had to determine the screen orientation programmatically and compose the widgets differently for landscape and portrait modes.

Error! Reference source not found. shows the size complexity of the Flutter code in terms of files and source lines of code (SLOC). We counted files (or libraries) instead of classes because of two reasons. Firstly, unlike Java, it is a common practice in Dart to have multiple classes in a single file, especially many private helper classes along with a few public classes. Secondly, most of our classes are stateful widget classes that consist of a pair of classes -- a widget class and a state class -- and it makes more sense to count the pair as one class. The Flutter implementation has 28 files and 4351 SLOC. It is 37% smaller than Android's 6949 SLOC (see Table 1).

Table 2. Code size of the MoQuiz Flutter app

Components	No. of Files	*Percent (%)*	No. of Lines	*Percent (%)*
Model	9	32.14	913	20.98
Network	1	3.57	229	5.26
User interface	18	64.29	3209	73.75

5. FINDINGS AND LESSONS LEARNED

As described in the previous section, we were able to reuse the design of the Android app when we rewrote it in Flutter. It is a good sign considering the platform differences, especially paradigm shifts such as reactive programming, isolate-based concurrency, and the user interface-as-code approach. We adopted the model-view-control (MVC) design of the Android app, and most coding efforts were spent on the view part. Rewriting the model classes in Dart was rather straightforward due to the similarity of collections APIs between Dart and Java. It was essentially a manual translation of constructs of one language to equivalent ones of another language. However, we had to do somewhat more work to convert the thread-based networking code to asynchronous functions, but the work resulted in cleaner code. One interesting aspect of the Java-to-Dart code translation is type safety. Java is a *type-safe* language in that an operation on an object is not allowed unless it is valid for that object. This of course is guaranteed statically through static type checking. Without much thought, we initially tried to preserve the type safety of Java statements as we translated them to Dart, if needed, by checking the type of an object at runtime. This often resulted in bloated code, especially when a library function returns an object of the *dynamic* type. In Dart, the *dynamic* keyword can be used as a type annotation to state that a variable's type is determined at runtime. We eventually gave up and had to be satisfied with even making some of our own functions to return objects of the dynamic type. Perhaps, at the beginning of the project, we should have given some thoughts on typing and decided on the level of type checking that we wanted in our Dart code, e.g., for public interfaces, static typing when possible, and dynamic typing when needed. Luckily, when we rewrote, or translated, Java code to Dart, there were not any surprising platform differences. However, we encountered one subtle difference in the libraries. In our Java code, we had an expression like name.replaceFirst("[^0-9]+", "") to replace a substring that matches a regular expression pattern, and without much thought, we copied it verbatim to the Dart. Later, after hours of frustrated debugging to fix a seemingly unrelated error, we learned that this expression was the root cause. In Dart, a regular expression pattern has to be written with the RegExp class, not as a bare string!

How does the Flutter implementation compare with that of the Android version in terms of the source code sizes? We learned that the Flutter app is about 37% smaller than the Android app in SLOC, 6949 lines of Java code vs. 4351 lines of Dart code. We believe this is due to both the Dart language and the Flutter framework. Dart provides more concise notations than Java along with language constructs and features like automatic generation of getters and setters. The Flutter APIs are relatively straightforward at a higher level of abstraction compared with those of Android. For example, navigating between screens (widgets) is surprisingly simple and can be done with a single framework method call, and both the parameters and the return value are type-checked. Navigating between screens (activities) on Android is a lot more involved with no type checking for parameters and the return value.

Regarding the code size, we also learned that the percentages of the user interface (UI) code over the whole code are similar between the two apps with 73% for Android and 74% for Flutter (see Figure 7). This is quite interesting considering the fact the Android UI-related code is really control code because the user interface layout is declared in XML. However, in the UI-as-code approach of Flutter, most code is indeed layout code concerned with defining widgets and composing them. We initially thought that Flutter would require a lot more user interface code than Android. However, the numbers show the opposite, i.e., more control code on Android. It may mean relatively more efforts on the Android user interface part -- designing a user interface in XML and writing its control code in Java.

Even if Flutter provides a similar set of widgets for common types of user interactions as Android does, e.g., text view, text edit, button, checkbox, icon, and list view, it took a bit of time and practice to be efficient with the UI-as-code approach. In particular, the "everything is a widget" principle was a small paradigm shift. In Flutter, even such things as padding, position, size, text style, theme, and gesture are indeed widgets. And you code a user interface by composing these widgets, resulting in a structure called a *widget tree*. Writing and editing deeply nested widget trees was painful even with a good editor. Some developers even call them the "nested hell". We quickly started to flatten and split the widget tree, e.g., by introducing private helper methods, to organize our code better. We do not know, however, if there are any performance implications in such a restructuring of a widget tree.

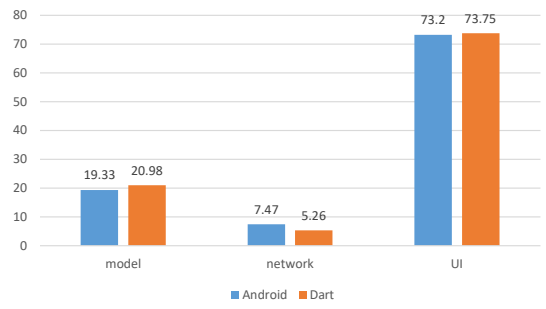

Figure 7. Distribution of source code

Another paradigm shift is the reactive programming model of Flutter. The user interface is *reactive* in that a widget is automatically rebuilt and redrawn when the state changes. This also means that one needs to manage the state that may change due to user interactions or background tasks. The state managed by a stateful widget is a local state for a single widget. So, one big question is: how to share the app state among multiple widget classes? For example, when a user selects a color in one widget class, another widget class that uses the selected color needs to be rebuilt and redrawn. Unfortunately, there is no official or standard approach for implementing this sort of a *global state*, a state shared by multiple (unrelated) widgets. There are numerous approaches proposed including Scoped Model, Business Logic Components (BLoC), Redux, and Mobx, some just a set of design principles and others providing Flutter packages (see http://fluttersamples.com for samples of different state management styles). Since Flutter is a relatively new technology, it requires hard work to search the best practices, especially for the beginning app developers. Nevertheless, the choice of state management is one of the most important design decisions, as it determines the overall architecture of the app. In our MoQuiz app, we adopted one of the simplest approaches or architectures, called the *scoped model*, but we are not sure whether another approach makes more sense for our app.

One common task for app developers is to release more than one version of the app to the market, e.g., free and paid versions. Even for our MoQuiz app, there are two versions, one for the instructor and the other for students. Android provides a way, called *build variants,* to maintain the different versions or flavors of an app in a single project. You can create several variants of your app by specifying which source code and resource files are to be included in a build. There is no such built-in support in Flutter.

We learned that the libraries included in the Flutter SDK are limited, missing many functionalities. For example, there is not even a library for showing notifications. Fortunately, there are a huge number of third-party libraries and APIs, called *packages* in Dart/Flutter, to access the underlying platform's features or services in a platform-neutral way. Thus, it is common in Flutter development that you use third-party libraries or packages. In most cases, however, there is a multitude of available packages contributed by different developers (see http://pub.dev/flutter/). For a simple task like displaying a toast – a short message in a small popup typically used to provide simple feedback about an operation – there are several dozens of packages available. One will likely become successful and be part of the Flutter SDK or de facto standard, but you don't know which one as you have to pick one for use in your app development. It should be also noted that some packages require you to perform platform-specific settings or configurations, which may require some level of knowledge on both Android SDK and iOS SDK. We used about 20 different open-source packages for our MoQuiz app. The API documents of several of these packages could be definitely improved.

6. RELATED WORK

Mobile application development practitioners today have to decide whether to develop two native apps – one for Android and the other for iOS – or to develop a cross-platform app, an app that is compatible with both Android and iOS. To reflect this industry need, there have been several studies on surveying, comparing, and evaluating cross-platform app development techniques, approaches, tools, and environments [5] [8] [11] [15] [19]. Since Flutter is a relatively new technology, it was not included in any of the existing studies. We also found no published work on rewriting Android apps in Flutter.

Platform differences are the primary source of challenges in developing cross-platform applications regardless of mobile or other platforms. It was shown, for example, the application programming interface (API) differences are the major factor that determines the amount of code reuse possible between Java and Android Java [4]. Even the two seemingly equivalent platforms of Java and Android Java have significant API differences, e.g., syntactic and semantic differences, built-in vs. third-party features, platform restrictions or constraints, and design guidelines [4]. These differences are of course present between two different languages such as Android Java and Dart (of Flutter). Ironically, programming language differences on the same API don't seem to be of much concern as evidenced by tool support such as Android Studio's automatic conversion of Java code to Kotlin. Our finding also confirms this in that we were able to rewrite the model code by essentially translating Java to Dart because Dart's collection API is very similar to the Java collection API. There is some existing work on testing platform differences, e.g., detecting inconsistencies in multi-platform apps [9], API compatibility issues of an Android app [10] [18], and inconsistencies of an Android app on different devices [6]. It would be interesting to adapt some of the work to the rewriting of Android apps in Flutter.

7. CONCLUSION

We rewrote an Android native app in Flutter to support both Android and iOS. We used an iterative approach to address platform differences with minimal impact as they show up during the development. There are several interesting findings and contributions made by our work. We identified platform differences between Android and Flutter in terms of languages, programming styles, and user interface design. The user interface part required most of our coding efforts. Rewriting the model part was rather straightforward even in the presence of language differences such as type checking and concurrency models. The resulting Flutter code is cleaner and about 37% smaller than the Android code: 4351 lines of Dart code vs. 6949 lines of Java code. Ironically, the percentages of the user interface code over the whole code are similar between the two versions at 73.20% and 73.75% for Android and Flutter, respectively. It might mean that a lot more control code of the user interface needs to be written on Android, as the layout is declared in XML.

So, should we start rewriting our Android apps in Flutter? One determining factor we think could be the maturity of the Flutter platform as well as its support tools and community. We noticed a lack of built-in, official, or sort of standard way of doing things in Flutter. They include not only available libraries and application programming interfaces but also guidelines, e.g., state management, externalization of app resources, the diversity of devices and user bases, and build variants. Another concern could be the availability of a large number of (premature) third-party libraries, or packages, to access the underlying platform's features or services in a platform-neutral way. We need to sort them out and pick one for our app. In Flutter, we use a lot of them unless we are willing to write our own platform-specific native code for both platforms. Therefore, it is very important to study carefully whether the Flutter platform and its tools support stably all the features needed by the apps to be rewritten.

REFERENCES

[1] *Android 10 Compatibility Definition*, accessed 24 August 2020, https://source.android.com/compatibility/10/android-10-cdd.

[2] A. Biorn-Hansen, C. Rieger, et al., An empirical investigation of performance overhead in corss-platform mobile development frameworks, *Empirical Software Engineering*, 25:2997-30240, Springer, June 2020.

[3] G. Bracha, *The Dart Programming Language,* Addison-Wesley, 2016.

[4] Y. Cheon, C. Chavez and U. Castro, Code reuse between Java and Android applications, *14th International Conference on Software Technologies* (*ICSOFT*), Prague, Czech Republic, July 26-28, 2019, pp. 246-253.

[5] I. Dalmasso, S. K. Datta, C. Bonnet and N. Nikaein, Survey, comparison and evaluation of cross platform mobile application development tools, *9th International Wireless Communications and Mobile Computing Conference (IWCMC)*, Sardinia, 2013, pp. 323-328/

[6] M. Fazzini and A. Orso, Automated cross-platform inconsistency detection for mobile apps, *IEEE/ACM International Conference on Automated Software Engineering* (*ASE*), Urbana, IL, 2017, pp. 308-318.

[7] *Flutter*, accessed 24 August 2020, https://flutter.dev/.

[8] 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.

[9] M. E. Joorabchi, M. Ali and A. Mesbah, Detecting inconsistencies in multi-platform mobile apps, *26th International Symposium on Software Reliability Engineering (ISSRE)*, Gaithersbury, MD, 2015, pp. 450-460.

[10] L. Li, et al., CiD: automating the detection of API-related compatibility issues in Android apps, *27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, July 2018, pp. 153-163.

[11] M. Martinez and S. Lecomte, Towards the Quality Improvement of Cross-Platform Mobile Applications, *4th IEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft),* Buenos Aires, 2017, pp. 184-188.

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

[13] 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.

[14] 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.

[15] 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, 2012.

[16] A. Ribeiro and A. R. da Silva, Survey on Cross-Platforms and Languages for Mobile Apps, *8th International Conference on the Quality of Information and Communications Technology*, Lisbon, 2012, pp. 255-260.

[17] 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.

[18] L. Wei, Y. Liu, and S. C. Cheung, Taming Android fragmentation: characterizing and detecting compatibility issues for Android apps, *31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, August 2016, pp. 226-237.

[19] S. Xanthopoulos and S. Xinogalos, A comparative analysis of cross-platform development approaches for mobile applications, *6th Balkan Conference in Informatics*, September 2013, pp. 213-220.