Design of Diary Applications for Vital Sign Registration Targeted at Multiple Android Application Stores

Ivan Shchitov, Eldar Mamedov, Ilya Paramonov
P.G. Demidov Yaroslavl State University
Yaroslavl, Russia
{ivan.shchitov, eldar.mamedov}@e-werest.org, ilya.paramonov@fruct.org

***Abstract*—The paper considers two aspects of expanding the user base of mobile applications for vital sign registration: making one application easily accessible from the others and targeting at multiple application stores. We provide a special design solution that allows to resolve these issues in a maintainable way.**

I. Problem Definition

A. Diaries and aggregators for vital sign registration

Application stores contain many medical applications. There are separate diaries, each for a particular vital sign, for example Blood Pressure[1], Weight Manager[2], Heart Rate Monitor[3]. There are also aggregators that allow to register several vital signs in one applications, such as myFitnessCompanion[4], TactioHealth[5], and Health-Tracker[6].

Applications of the first category are highly specialized and can be easily found by patients interested in registration of the particular vital sign (e.g., blood pressure or glucose level), but they help a little when the user needs to track several vital signs. Aggregators allow to register multiple vital signs at once but their functionality often is not so rich and a user who needs to track only blood pressure can hardly find them.

In this paper we consider the problem of combining these two approaches to employ benefits of the both. We achieve it by providing several highly specialized applications that are specifically interconnected that allows the user to treat them as a single application.

B. Publishing in different application stores

There are many different Android application stores: Google Play, Samsung Apps[7], Amazon Mobile App Distribution[8], Opera Apps[9], Nokia Store[10] etc. Each store covers its own user base making developers interested in bringing their applications to many of them. Unfortunately applications targeted at one store are not suitable for other stores because each of them has its own specific APIs and services, for example, in-app purchases mechanism, advertising services, etc.

Therefore, it would be beneficial for developers to design a mechanism that allows to prepare the application binaries for publication in different stores in accordance to requirements of these stores.

II. Companion Applications

A. Overview of companion applications

We consider our solution in a case study of three applications developed by EwerestMD LLC: Blood Pressure Diary[11], Weight Diary[12] and Healthy Sleep Diary[13].

Blood Pressure Diary is an application for tracking and analyzing blood pressure measurements. It helps people that suffer from various diseases of the blood circulatory system, e.g., hypertension or hypotension. Weight Diary is an application for tracking and analyzing body weight measurements.

These applications provide functionality of storage and intelligent management of measurements: addition, displaying, modification and deletion of measurements; statistic reports on measurement including maximum, minimum and average for given time periods (morning, day, evening, night); classification of blood pressure measurement based on the WHO / ISH Hypertension guidelines [1]; import and export of measurement data and relevant medical notes for physicians and other medical systems.

Healthy Sleep Diary is a simpler application for tracking sleep time. It motivates people to go to bed early with the use of entering sleep time and tracking estimated and actual sleep time. The application also provides capabilities of setting the personal sleep norm, selecting a method for tracking sleep time, etc.

We refer Blood Pressure Diary, Weight Diary, and Healthy Sleep Diary as “companion applications” because they are based on similar principles, have similar user interface, and integrated to seem like a single application as it is shown in the following subsection.

[1]https://play.google.com/store/apps/details?id=com.freshware.bloodpressure
[2]https://play.google.com/store/apps/details?id=weight.manager
[3]https://play.google.com/store/apps/details?id=com.heartrate.monitor
[4]https://play.google.com/store/apps/details?id=com.myfitnesscompanion
[5]https://play.google.com/store/apps/details?id=com.tactio.tactiohealth
[6]https://play.google.com/store/apps/details?id=com.benoved.phr_lite
[7]http://http://apps.samsung.com
[8]http://www.amazon.com/mobile-apps/b?node=2350149011
[9]http://apps.opera.com
[10]http://store.ovi.com
[11]https://play.google.com/store/apps/details?id=org.fruct.yar.bloodpressurediary
[12]https://play.google.com/store/apps/details?id=org.fruct.yar.weightdiary
[13]https://play.google.com/store/apps/details?id=org.fruct.yar.healthysleepdiary

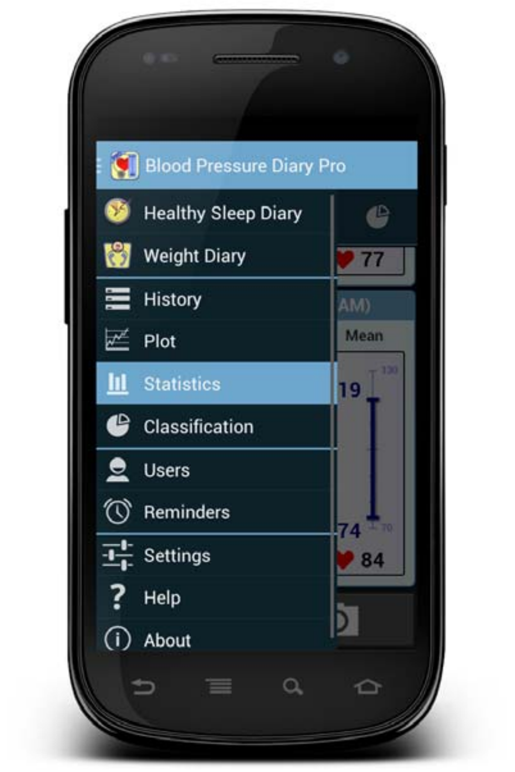

Fig. 1. Navigation drawer of the Blood Pressure Diary application

B. *Access to companion applications from navigation drawer*

In order to provide quick access to companion applications to the user we employ activities of the Android platform.

An activity is an application component that interacts with user. The activity is usually responsible for one function, such as dialing the phone, taking a photo, sending an SMS, or viewing a map. Activities in Android have a remarkable feature: an activity can invoke another activity even if these two activities belong to different applications. In our case, this makes activities suitable for connecting several vital sign diary applications into the whole.

For the user this connection looks as follows. Each application has a navigation drawer—a panel that can be pulled-out from the left edge of the screen and displays main navigation options of the application. In the drawer each companion application has its own item. For example, navigation drawer of Blood Pressure Diary contains “Weight Diary” and “Healthy Sleep Diary” items (see Fig. 1). When the user taps one of these items the main activity of the corresponding companion application gets started.

Usually when an activity is started from another activity, both the activities are kept in the activity stack allowing to return the former activity by pressing the “Back” button. But in our case the starter activity is removed from the activity stack to mimic working with a single application. The Android navigation guidelines [2] state that “Back” button should return to the previous screen but we treat them as the same screen that provides access to different measurements.

If the companion application is not installed and the user taps the corresponding item in the navigation drawer the initial application proposes to install the ordered companion application from an application store.

III. Targeting at multiple application stores

A. *Differences between assemblies for different stores*

Each particular application store imposes its own requirements on applications published in this store. For example, the application can be restricted to use only specific APIs for particular tasks (e.g., maps, advertisement, in-app purchase and so on). Another example could be prohibition to reference contents from stores other than one from which it has been installed.

To meet these requirements developers need to separate assembly of appliations that will have the following differences:

- different set of libraries;
- differences in the manifestos;
- differences in the code.

For example, in-app purchase mechanism of the Google Play requires addition of the In-App Billing library to project, billing permission to manifest, creation and use of the IabHelper class instance. But in-app purchase mechanism of the Samsung Apps requires addition of the Samsung IAP SDK to project, billing, internet permissions and InboxActivity, PaymentActivity, ItemActivity to manifest, creation and use of SamsungIapHelper class instance.

Thus, we cannot use both libraries at the same time and choose what activity should be started, as they are to be registered in the same manifest. The solution is to make separate application assembly for each store.

B. *Project build types*

To build our applications we use Gradle [3] and Android plugin for Gradle [4]. It allows to solve the problem of separate assemblies for different application stores in two ways: using build types and flavors.

We prefer build types over flavors because for each flavor at least two assemblies are created: “assemble*FlavorName*Debug” and “assemble*FlavorName*Release”. But we do not need to create multiple debug assemblies for a single application because the debug is not connected to any specific application store. We would like to create a single assembly for debugging and a several release assemblies for different application stores. For this purpose build types are more suitable.

For each build type we can create a separate package with sources specific for each application store. It allows to create different manifests and employ different classes and constructions for different assemblies. At build time the source code and resources of the build type are used in the following way: the manifest from the folder of the corresponding build type is merged into the application manifest, the sources acts

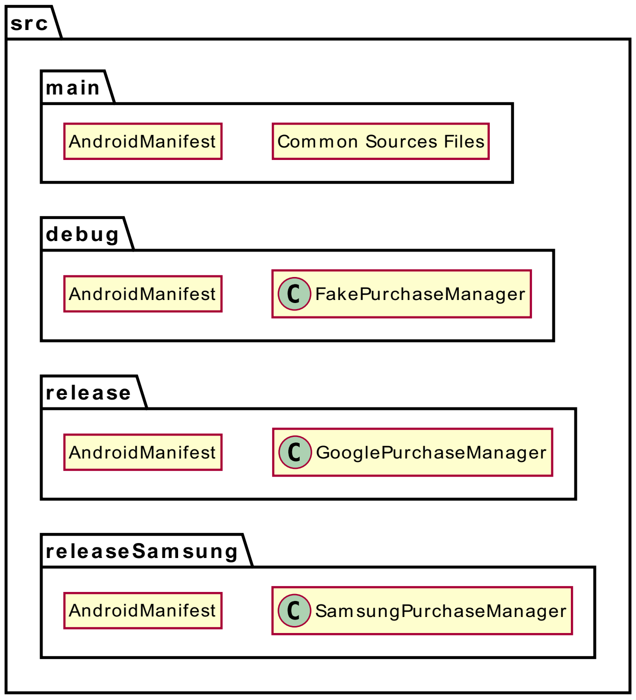

Fig. 2. The directory hierarchy of the project

as just another source folder, and the resources override the main resources, replacing existing values.

For each build type, a new "assemble*BuildTypeName*" task is created.

For each of our applications we have three build types: "debug", "release" and "releaseSamsung". Therefore we have "assembleDebug", "assempleRelease", and "assembleReleaseSamsung" tasks to debug application and to build binaries for Google Play and Samsung Apps respectively.

For each build type we create package in location src/*build type name*/ (see Figure 2). For "debug" build type it is "debug" package, for "release" build type it is "release" package and for "releaseSamsung" build type it is "releaseSamsung" package. We placed different manifests, configuration files and sources code to different packages. Thus, we can use different implementations of different classes for different builds.

C. Unified interfaces for store-specific classes

We consider using the build types on the in-app purchase mechanism implementation for two stores: Google Play and Samsung Apps. They provide different classes for in-app purchase operations. API for Google Play provides IabHelper class, but API for Samsung Apps provides SamsungIapHelper class. These classes have different implementations, dependencies, and (the most important) different interfaces. That is why we need to unify interface for these classes to avoid fragmentation of the code base.

To resolve this issue we designed the abstract PurchaseManager class shown in Figure 3. It has a constructor, where we check whether the application is purchased or not, the startPurchase() abstract method for starting purchase flow, and

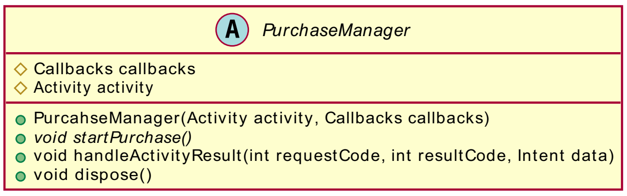
A
PurchaseManager
Callbacks callbacks
Activity activity
PurcahseManager(Activity activity, Callbacks callbacks)
void startPurchase()
void handleActivityResult(int requestCode, int resultCode, Intent data)
void dispose()

Fig. 3. The interface of the abstract PurchaseManager class

the dispose() method for destroying of purchase helper class instance. The callbacks object is used for handling the purchase result in other parts of the application.

We created GooglePurchaseManager, SamsungPurchaseManager and FakePurchaseManager classes that are inherited from the PurchaseManager class. Constructors of subclasses invoke constructor of the superclass and, if application has not been purchased, create the purchase helper class instance (IabHelper for Google Play and SamsungIapHelper for Samsung Apps). FakePurchaseManager class is placed in the "debug" package and needed for the debugging purposes, it does not use any in-app purchase API. GooglePurchaseManager and SamsungPurchaseManager are placed in "release" and "releaseSamsung" packages accordingly.

To instantiate a proper PurchaseManager implementation we define fully qualified name of the purchase manager class in metadata of the manifest for each build type. In runtime we extract the class name from the manifest and create the proper object via reflection.

This solution can be easily extended to support more specific APIs and application stores.

D. Navigation drawer configuration

Usually application stores prohibit using links to other stores in applications published in these stores. It makes sense, because the store can only be sure, that it services are installed on the user's phone, but it creates troubles when publishing applications that contain links to other applications in their interface.

In order to avoid changes in the code targeted at different application stores we moved links control to a configuration file. Configuration file is an XML file with one top-level "apps" element containing several "app" elements. Each "app" element contains attributes with following data regarding a companion application: an id, an application name, a package name, a description, an icon, a link to the application in the store, a link to the web page devoted to the application in the store.

Each assembly has its own version of the configuration file, which has links to the application specific for a particular application store. For example, the "release" assembly contains the file with links to the application page in Google Play Market, whereas the "releaseSamsung" assembly contains the file with links to the application in the Samsung Apps store.

Such an approach is convenient for developers as it allows to make changes in navigation drawer contents (e.g., to add a new companion application) leaving the source code of the application intact.

IV. Conclusion

In the paper we developed the concept of the "companion application" and implemented it for three vital sign diary applications for Android platform. It simplifies finding appropriate diary for vital sign registration for users and allows them to extend their functionality by means of other diaries. For developers such an approach allows to expand the user base.

Also we developed a mechanism that allows to substitute implementation of functionality depending on the application store at which the application is targeted.

All solutions described above are universal and can be applied to many interconnected applications and many application stores.

Acknowledgements

The authors would like to thank Sergey Balandin for his initial idea of companion applications and subsequent fruitful discussions on this topic.

This work was partially supported by grant No. 14-01-31539 from the Russian Foundation for Basic Research (RFBR).

References

[1] Guidelines Subcommittee, "1999 World Health Organization—International Society of Hypertension. Guidelines for the Management of Hypertension," *in J. hypertension*, 1999;17:151-183.

[2] Navigation with Back and Up: Android Developers, Web: http://developer.android.com/design/index.html.

[3] B. Mushko, *Gradle in Action*, Shelter Island, NY: Manning Publication Co., 2014.

[4] Gradle Plugin User Guide — Android Tools Project Site, Web: http://tools.android.com/tech-docs/new-build-system/user-guide.