CS31620 Vocabulary App - Project Report

Rhys Evans

Department of Computer Science,
Aberystwyth University,
Aberystwyth
rhe24@aber.ac.uk

Abstract. The basis of this project was to design and create an Android app to help users learn a new language vocabulary. The App I created is named 'Language Vocabulary Assistant', or 'LVA' in short. The key features of the app are: Specifying a primary and secondary language, storing words or phrases in a viewable list, testing knowledge of the stored words or phrases through practice, and reviewing performance in practices.

1 Design

This section will discuss all aspects of the App's design, this includes both the user interface design and the architectural design.

1.1 Architectural Design

Program Flow The app implements an event driven model, based mostly on user interaction. Most operations are invoked as a direct result of user interaction, therefore the majority of the app's operations are done in the UI thread. Because of this, the classes that do most of the heavy lifting are the Activity and Fragment classes, with the most important class being LVAMainActivity. This is the main activity and entry point of the app, and for the most part remains active throughout the app's lifecycle. The other two activities within the app are LVASetupActivity and PracticeActivity, together, these three activities encapsulate the individual features of the app well and guides program flow. The 'Setup' Activity handles language preference input, whilst the 'Main' activity handles the vocabulary entries and practice stats overview. The 'Practice' activity is dedicated to the quiz-style practice.

Package Structure There are three main packages in the app: **datasource**, **model** and **ui**. Within the **model** and **ui** packages are two sub-packages: **practice** and **vocabulary**. As the app's key features are to store and manage a vocabulary list and to use it for practice, this seperation felt reasonable. The **datasource** package contains the classes needed to implement the Room Persistence Library, with the exception of the entity classes, these are located in the **model** package.

Design Patterns The app loosesly follows the Model View View-Model design pattern. There are two main view models in the app: VocabularyListViewModel and PracticeViewModel, these are used to interact with the backend database (model) and update the UI (view). For example, the PracticeViewModel is able to access the PracticeAttempts model to display information in the PracticeOverview view. However, the app does violate the MVVM design pattern in some aspects. The MVVM pattern enforces a one to one relationship between a view and a view-model, however some views within the app have access to two view-models, thus violating the one to one relationship. For example, the PracticeActivity needs information from the vocabulary model in order to display the correct 'questions', however it also needs access to the practice model in order to store practice attempts. Given the relatively low complexity of the app's logic I felt this was a reasonable trade-off, however given the opportunity to refactor I would likely take an approach that would lead to lower coupling between the practice and vocabulary modules.

Data Persistence Data is persisted in two ways within the app: shared preferences and a backend SQLite database. The user's language selections are stored in the shared preferences, as they can easily be represented by a key-value pair there is no need to further complicate the database schema. The 'SharedPreferenceManager' is a singleton class intended to provide the app with a clean interface to the shared preferences, it handles all insertions, deletions and updates to the shared preferences. Each of the app's view-models has an instance of the shared preference manager, this provides a layer of abstraction between the view and the model.

The SQLite database stores all vocabulary entries and practice attempts in their designated tables, the database interfaces with the SQLite API using the Room Persistence Library. Therefore, for each entity there needs to be an entity class, a data access object class and a view-model class. The vocabulary entries entity encapsulates an entry in the vocabulary list, therefore it stores the word/phrase in the primary language and its translation in the secondary language. The practice attempt entity encapsulates an individual practice attempt, this enables the user to view their practice statistics and track performance. The practice attempt entity stores the date of a given attempt, the score achieved and the maximum score of the attempt.

1.2 User Interface Design

The app's UI design attempts to conform to the principles laid out in google's material design guidelines [1], using a tab layout and viewpager in the main activity encourages the idea of motion and coherent transformations. Dialogs are used to display important information as their layer of depth inherently imply importance. Dialogs are especially useful because they do not take the user away from the underlying view, but rather put it on pause whilst an action is requried.

The app's color scheme was chosen using google's color picker tool [2], the color scheme of the app uses a primary color with a light and dark variant and a secondary color used to style accents. Having variance in colour allows seperation between important UI elements and layout surfaces. The use of material design icons and android layouts in the app help provide an intuitive affordance, the most prominent example of this is the floating action button on the main activity.

The UI went through a few stages of prototyping before it was implemented fully, this accommodated the opportunity to easily try several different color schemes and layouts before committing to a full implementation.

2 Testing

In order to effectively test the app and ensure quality, a test table was created. Whenever a new feature was added a test would be designed to ensure it performed as expected. After implementation, each feature was manually tested on two seperate devices according to the criteria laid out in the test table. After all

features were fully implemented and had been manually tested, unit tests and UI tests were created to replace the manual tests for a select few features. As an improvement for future projects it would be very beneficial if the unit/UI tests were created at the same time as the test table entry is created, thus eliminating the need for labour-intensive manual testing. This could be acheived by refactoring the codebase early on to accommodate things like in-memory databases and Espresso-friendly views. Using unit/UI tests from the beginning would also improve support for regression testing to ensure reliability throughout the program when a new feature is added.

Overall, the manual testing that was done throughout the implementation process was very effective in highlighting issues in the code, maintaining an up-to-date test table certainly played a large role in this.

3 Review

References

1. Material Design. (2018). Introduction. [online] Available at: https://material.io/design/introduction/# [Accessed 5 Dec. 2018].
2. Color Tool - Material Design. (2018). Color Tool - Material Design. [online] Available at: https://material.io/tools/color/#!/?view.left=0&view.right=0 [Accessed 5 Dec. 2018].