Tingle Version 5

Thor V.A.N. Olesen

April 23, 2016

Contents

1 Design choices

This chapter outlines some of the design choices that have been made throughout the course and iterative extension of the Tingle Application project. In particular, the MVC pattern and SOLID principles will be used to exemplify how parts of the Android project has been designed. Also, some of the main features including searching, sorting, camera, persistent storage and barcode lookup will be outlined as part of the major focus in the final version of the Tingle application.

1.1 MVC

Firstly, I have separated the concerns of the software functionality by using the Model-View-Controller (MVC) design pattern (see package diagram in appendix A.1. The Views are composed of the XML layout files in the res layout package. Secondly, the Controllers are composed of the fragments, activities and widgets in the Controller package used to control the functionality of the views. Thirdly, the models are composed of the object-oriented domain classes in the Model package used to represent 'Thing' objects and hold the repository that defines how the application retrieves data from these objects. The MVC pattern has been used to avoid overlapping between the different functionalities of the program and thus achieve an overall modular design. Finally, yet another 'Helper' package has been used to hold classes that assist in providing some functionality that is not part of the overall MVC structure in the application (e.g. network, search and database functionality).

1.2 SOLID principles

SOLID (single responsibility, open-closed, Liskov substitution, interface segregation and dependency inversion) is an acronym used in programming to describe five principles used within object-oriented programming and design in order to create a system that is easy to maintain and extend over time.[1] Some of the guidelines have been applied in this project to make the source code clear extensible, which is exemplified on the next page.

1.2.1 Single Responsibility

The single responsibility principle states that 'a class should only have one reason to change'. An example of this in the project would be the RecyclerView in 'ThingListFragment' and its adapter 'ThingAdapter'. The adapter is used to take the Thing data from the database and adapt it to a view in the RecyclerView. The adapter only has one responsibility which is to map a Thing object to its corresponding view that will be displayed on the screen within the list of items.

[1] `https://en.wikipedia.org/wiki/SOLID_(object-oriented_design)` seen 23 April, 2016.

The onBindViewHolder method in the adapter is only responsible for the mapping from a Thing object to the view and does not perform any additional calculations. In this way, the Thing object and its calculation logic is not coupled to the adapter and we do not have to replicate any logic to display the Thing object elsewhere.

1.2.2 Open-closed

The open-closed principle states that software entities (classes, modules, functions) should be open for extension but closed for modification. Basically, the existing code should not have to be changed every time the requirements change. An example of this would be the GenericSort abstract class and ISort interface used in the Search package. The GenericSort class outline the common features that all sorting algorithms should have (exchanging and comparing items) and the sorting calculation is abstracted away into an ISort interface with one responsibility defined - to sort Thing objects alphabetically based on their name or location. If you want to change the way items are sorted you simple extend (inherit) from GenericSort and implement the ISort interface to override the sorting method. As a result, duplicated code is avoided and the GenericSort class is kept open for extension but closed for modification.

1.2.3 Liskov Substitution

The Liskov Substitution principle states that the objects in a program should be replaceable with instances of their subtypes without altering the correctness of the program. By way of example, the Thing Repository needs a list of Things when retrieving data from the database in the getThings() method. The repository only requires that the list of things is of type List¡Thing¿. However, when we call the method we provide an ArrayList¡Thing¿. Since ArrayList¡Thing¿ is a subtype of List¡Thing¿ we are replacing the instance of the requested type (List¡Thing¿) with an instance of its subtype (ArrayList¡Thing¿). In other words, the code depends upon an abstraction and the program will run without any issues since the repository depends upon the contract provided by the List interface. In short, we can replace any list data structure that extends List and not break the program.

1.2.4 Interface Segregation

The Interface Segregation principle states that the client should not be forced to depend upon interfaces that they don't use. This principle is sometimes violated in the project when implementing listeners on widgets used in the application UI. By way of example, the ViewPager class 'ThingPagerActivity' implements the onPageScrolled, onPageSelected and onPageScrollStateChanged listeners but only the first two of them are actually used. This leaves us with a 'fat interface' since we implement the full interface and provide dummy methods. Alternatively, one should provide many smaller interfaces to serve submodules in the project and avoid content in the interfaces that is not used.

1.2.5 Dependency Inversion

The last Dependency Inversion principle states that the software entities should depend on abstractions rather than concretions. In other words, abstractions should not depend on details and details should depend upon abstractions. By way of example, an abstraction could be the IRepository interface used in the model package to outline the CRUD operations that the Thing Repository should support for retrieving data from the database. The direction of the dependency is 'inverted' by using interface contracts. In this case, the IRepository interface will reverse the direction of the data layer dependency and is instead put in the 'business layer'. The business objects only interact with the database via the high level interface. The data layer implements the repository interface it contains concrete repositories which actually talk to the database. The dependency inversion maks unit testing possible since the domain objects now only depend on the IRepository interface that can be mocked or replaced with in-memory fake repositories. This has been violated to some degree in the project since a concrete Thing Repository is used in the controller classes to retrieve data.

1.3 Fragments and Communication

Fragments have been used to represent contents of the main page and list page UI since this could not be encapsulated to one activity UI component alone. For instance, the main page changes depending on the hardware configuration but this is not affected since each part of the UI is an independent fragment. Each fragment knows what they need to show and they do not communicate directly to each other. Instead, the fragments, 'TingleFragment' and 'ThingListFragment' communicate with their host activity, TingleActivity using a listener interface in the fragments that is implemented by the activity. Also, the fragment UI components are self-contained, modular and define their own layout and behaviour to allow reuse. The fragment captures the interface implementation during its onAttach() lifecycle method and can then call the interface methods in order to communicate with the activity. As opposed to using intents like in Tingle V3 where only activities were used. During runtime, the TingleActivity will receive events triggered from calling the interface method in the fragments (e.g. going back or showing the list of items). As a result, all Fragment-to-Fragment communication is done through the associated Activity and not directly.

1.4 Naming Conventions and Resources

Some of the conventions outlined in the book have been followed to uphold the guidelines within Android development. Specifically, the UI fields are denoted with 'm' naming conventions (e.g. "mButton") and string resource files are encapsulated in a separate XML file to avoid hard coding strings in the code (easy localization and cross language support).

1.5 New features

Some of the new features introduced in the new version of Tingle 5 include search and sorting, camera pictures, barcode lookup and persistent storage in SQLite. All the functionalities have been implemented within classes in the Helper package. Specifically, the Search package holds the classes described previously in SOLID chapter and is used to allow the user to search and sort items based on their names and locations with the exception of the date that cannot be used to sort even though it is displayed as a menu item. The Database package contains new classes used to retrieve data from a SQLite database and uses a DatabaseManager class to create a thread-safe connection. Finally, the network package contains classes used to fetch data from item barcodes over the network using the Outpan API and AsyncTask to ensure that it happens on a background thread without affecting the UI. The camera functionality has been implemented using implicit intents to declare a general action to be performed allowing the built-in camera in the phone to be used to take a picture and register it with an item. Same goes for the barcode scanner that fires up the ZXing Barcode Scanner via an Intent. All together, these new features have been implemented to meet the minimum requirements of the final application.

2 User Interface

The User Interface is composed of the many XML layout files in the res layout package and include a layout for both portrait and landscape screen orientation. In this regard, I have chosen to make different layout files based on the orientation because some of the widgets are not relevant. By way of example, I do not include a Back button in the user interface composed in landscape mode. This is based on the fact that the main page is already shown in the fragment of the left container. Also, I do not include the "See Items" button used to redirect to the list of items in the ListFragment because they are already shown in landscape mode. Thus, I have designed the user interface for supporting both portrait and landscape orientation by using different layouts and widgets.

3 Testing

3.1 Manual Testing

I have tested the application manually in an emulator on my computer and on a real Android device by adding things, flipping the phone, deleting things from the list, switching pages, scanning item barcodes and taking pictures. Manual testing was easy to do and gave me fast qualitative information on how my app behaved in different work flows. Specifically, I tested the usability (user experience), interface (testing of buttons) and compatibility (different screen sizes and orientations). Also, I have made some unit tests of the Model package to clarify that items are stored correctly and the underlying data logic layer

works. I have chosen not to look into automated UI tests due to the relatively small complexity of the application so far. The manual testing combined with the unit tests of the model and repository allowed me to test all aspects of my app. However, I considered using the unit test framework 'Roboelectric' to remove the Android SDK and emulator dependency that makes running tests slow.

3.2 Test Sequences

Below are some of the test sequences that have been used while manually testing the app. The test sequences cover important functionalities of the app (including error handling) and involve all parts of the user interface. Five test sequences have been included to cover the most essential use cases and functions.

Use Case	Test Sequence	Result
Register thing	Start app in portrait mode.	Worked
	Fill name and location in text fields.	Worked
	Click scan button to register barcode.	Worked
	Fetch name from scanning item.	Worked
	Store thing information in SQLite database.	Worked
	Check that new thing is stored correctly.	Worked

Use Case	Test Sequence	Result
Search thing	Start app in portrait mode.	Worked
	Navigate to list of items by clicking 'See Items'.	Worked
	Click on search menu item.	Worked
	Type in search string and check for one hit.	Worked
	Type in search string and check for multiple hits.	Worked
	Type in search string and check for no hits.	Worked

Use Case	Test Sequence	Result
Delete thing	Start app in portrait mode.	Worked
	Navigate to list of items by clicking 'See Items'.	Worked
	Long-tap item to activate multi selection.	Worked
	Select one item and delete.	Worked
	Select multiples items and delete.	Worked
	Try to delete item without long tap and multi select.	No delete
	Check that list is updated correctly.	Worked
	Check if item(s) are deleted from database.	Worked

Use Case	Test Sequence	Result
Scan barcode	Start app in portrait mode.	Worked
	Click 'Scan Item' and scan barcode.	Worked
	Check if barcode is inserted in barcode field.	Worked
	Wait for item name to be fetched from internet.	Worked
	Click 'Add New Thing'.	Worked
	Check if thing information is stored in database.	Worked

Use Case	Test Sequence	Result
Register Thing picture	Start app in portrait mode.	Worked
	Navigate to list of items by clicking 'See Items'.	Worked
	Navigate to item detail page by tapping item.	Worked
	Click camera button and take picture.	Work (not root)
	Check picture in image view and store item.	Worked
	Check if thing information is stored in database.	Worked

4 Problems

The main challenges in the Tingle version 4 were to replace the activities by fragments, associate the fragments through interfaces and most importantly to support different layouts based on the orientation of the device. A lot of time was spent on implementing the interfaces used to communicate between the fragments and the host activity while also moving the layout inflation in the onCreate() methods to the onCreateView() methods used in the fragments. Also, I had to make some deliberate choices about what to show on the screen based on the orientation (e.g. the back button was previously not shown in landscape mode). As opposed to Tingle version 4, a Scroll View has been used to fit all of the UI components in the detailed page and the landscape main page. This has allowed me to disregard the need to choose the most important UI components to be shown on the screen. Finally, one of the biggest problems introduced by the new features in version 5 derived from the searching and sorting functionality. The new functionality made it hard to update only specific parts of the list using notifyItemChanged(i) in the RecyclerView Adapter. By way of example, when the list of items is sorted all items may potentially swap positions and thus require the whole list to refresh using notifyDataSetChanged(). This also introduced an annoying bug when trying to delete items in a sorted list where items had changed their ViewHolder position, ultimately requiring a full refresh of the listcontent. Thus, the new version of Tingle has been designed to trade off some performance in return for the new search and sorting functionalities that allow the user to search and sort items based on their name and location. However, the extent of the list size has been judged to be relatively small with no real performance overhead added upon these changes.

5 References

[1] Bill Phillips, Chris Stewart, Brian Hardy and Kristin Marsicano. *Android Programming: The Big Nerd Ranch Guide*. Big Nerd Ranch, Reading, LLC, 2015.

[2] Robert Sedgewick and Kevin Wayne. *Algorithms, 4th Edition*. Princeton University, Addison-Wesley, 2011.

[3] Android Developer: Communicating With Other Fragments, retrieved on April 23, 2016 from `http://developer.android.com/training/basics/fragments/communicating.html`

[4] Android Developer: Connecting To The Network, retrieved on April 23, 2016 from `http://developer.android.com/training/basics/network-ops/connecting.html`

[5] Android Developer: Processes and Threads, retrieved on April 23, 2016 from `http://developer.android.com/guide/components/processes-and-threads.html`

[6] Android Developer: Services, retrieved on April 23, 2016 from `http://developer.android.com/guide/components/services.html`

[7] Android Developer: Loading Large Bitmaps Efficientlys, retrieved on April 23, 2016 from `http://developer.android.com/training/displaying-bitmaps/load-bitmap.html`

[8] Vogella: Android Intents, *Vogella*, 2014 Lars Vogel, retrieved on April 23, 2016 from `http://www.vogella.com/tutorials/AndroidIntent/article.html`

[9] Big Nerd Ranch: Testing the Android way, *Big Nerd Ranch*, 2016 Josh Skeen, retrieved on April 23, 2016 from `https://www.bignerdranch.com/blog/testing-the-android-way/`

[10] Big Nerd Ranch: RecyclerView MultiSelect, *The MIT Licence*, 2014 Big Nerd Ranch retrieved on April 23, 2016 from `http://bignerdranch.github.io/recyclerview-multiselect/`

[11] Princeton University: SelectionSort Source Code, *The MIT Licence*, 2010 Robert Sedgewick and Kevin Wayne, retrieved on April 23, 2016 from `http://algs4.cs.princeton.edu/21elementary/Selection.java.html`

[12] ZXing Barcode Scanner, *Apache License*, 2004 RZXing, retrieved on April 23, 2016 from `https://github.com/zxing/zxing`

A Appendices

A.1 Tingle Package Diagram

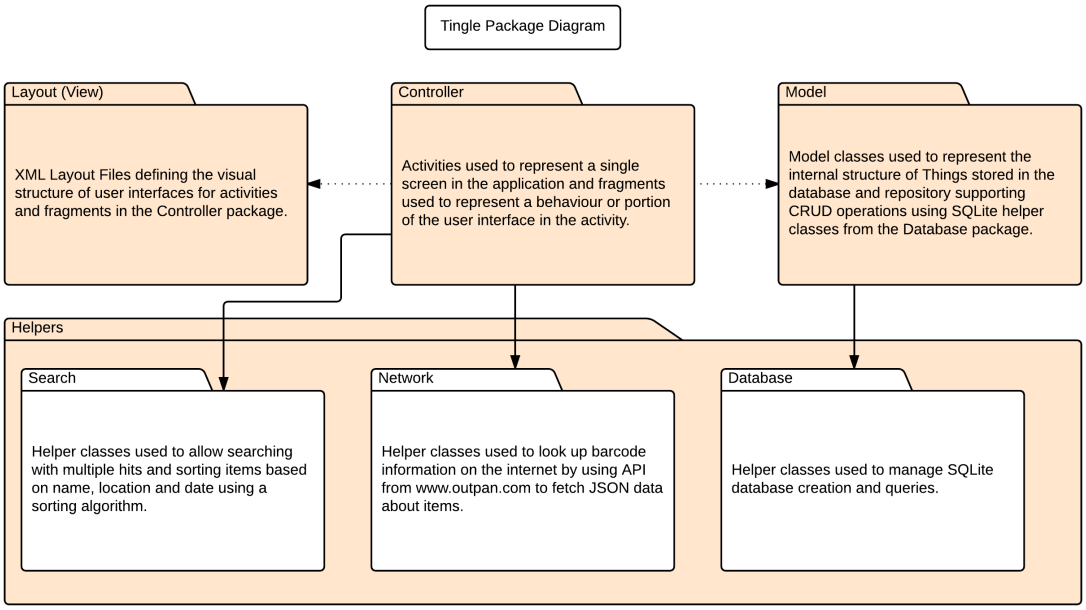

Figure 1: UML Package Diagram

A.2 Tingle View Class Diagram

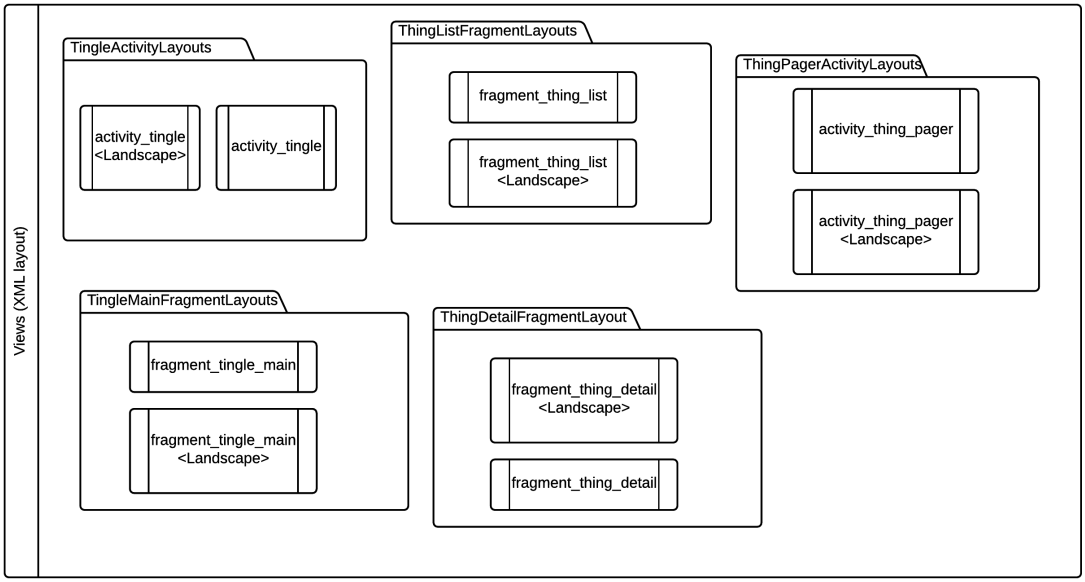

Figure 2: UML View Class Diagram

A.3 Tingle Model Class Diagram

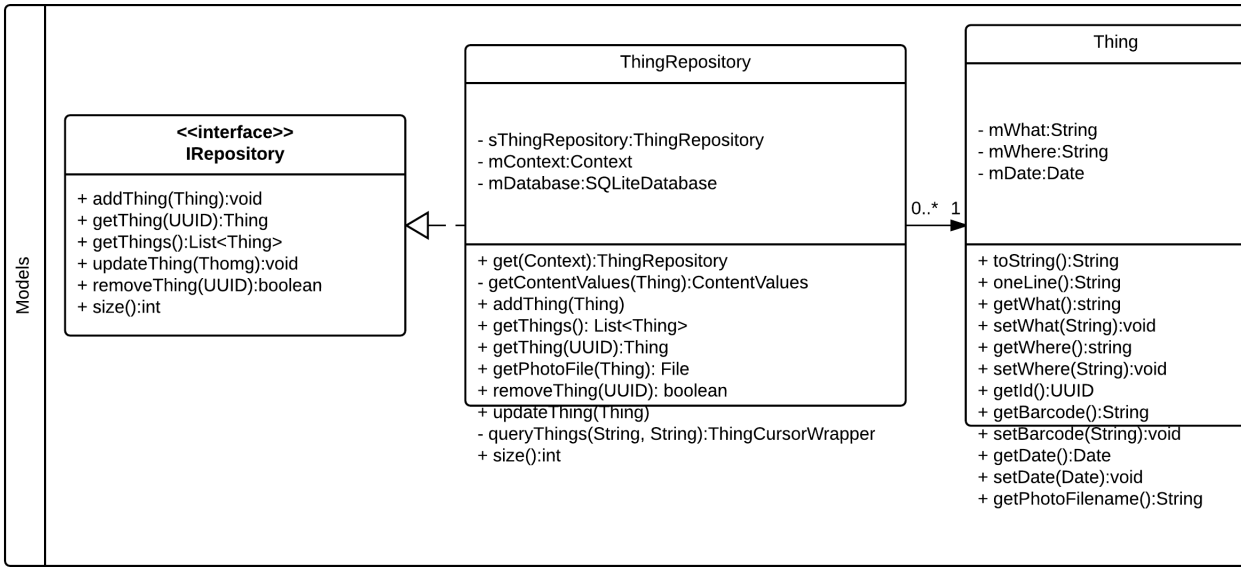

Figure 3: UML Model Class Diagram

A.4 Tingle Controller Class Diagram

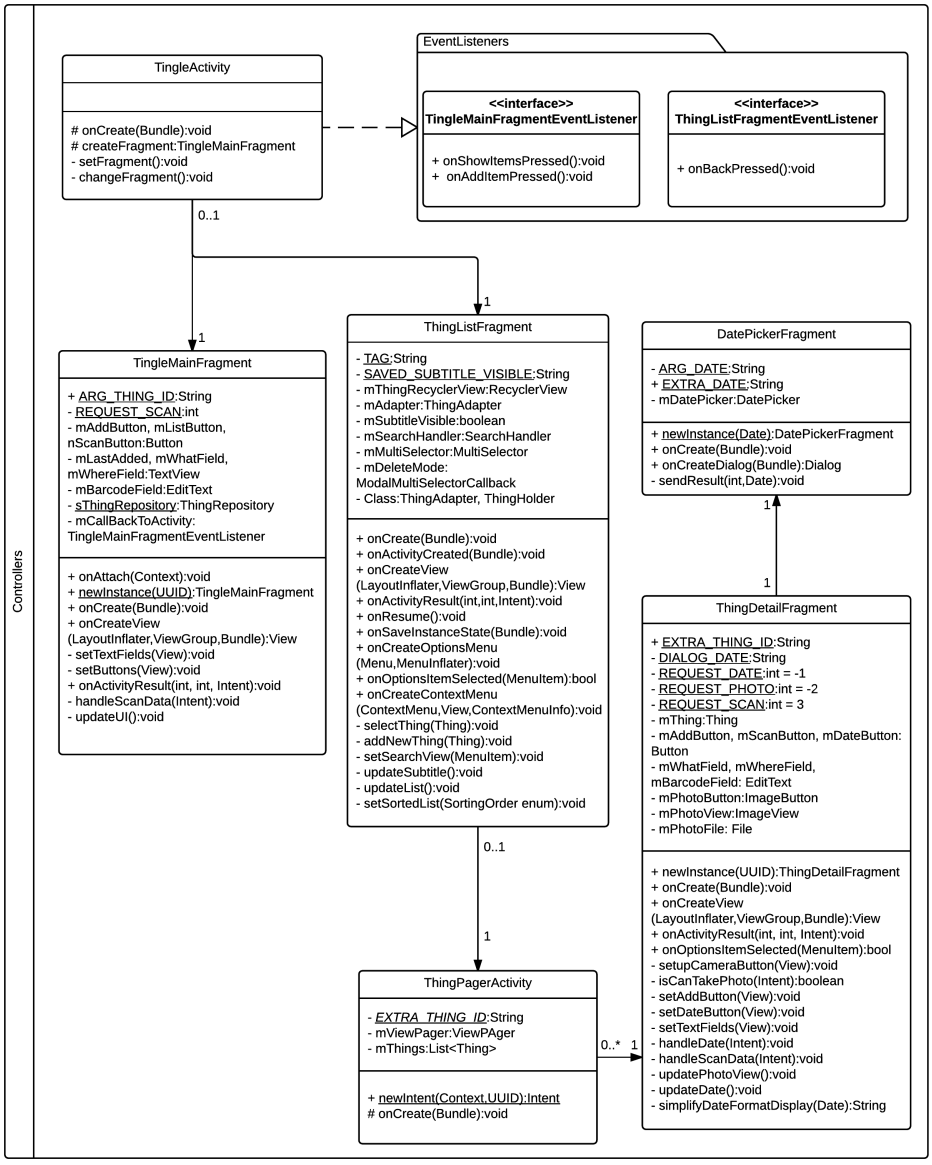

Figure 4: UML Controller Class Diagram

A.5 Tingle Component Subsystem Diagram

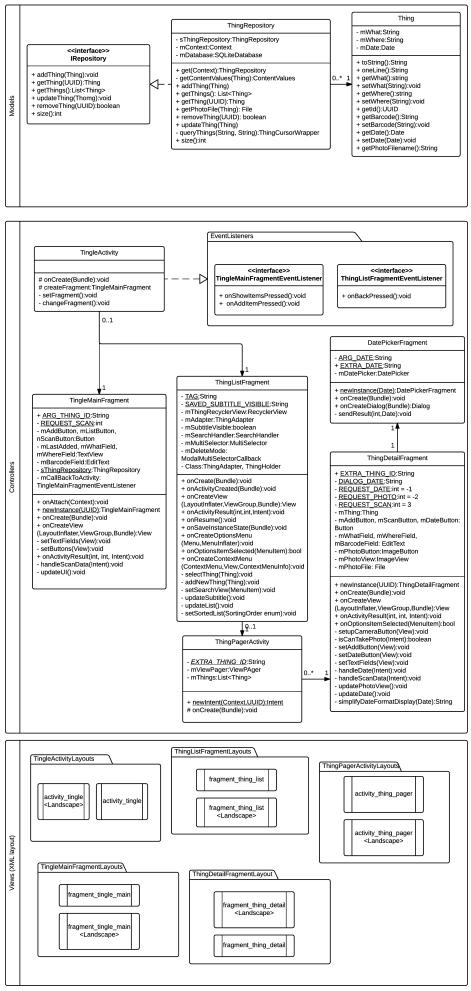

Figure 5: UML Component Subsystem Diagram

A.6 Tingle App Main Page UI

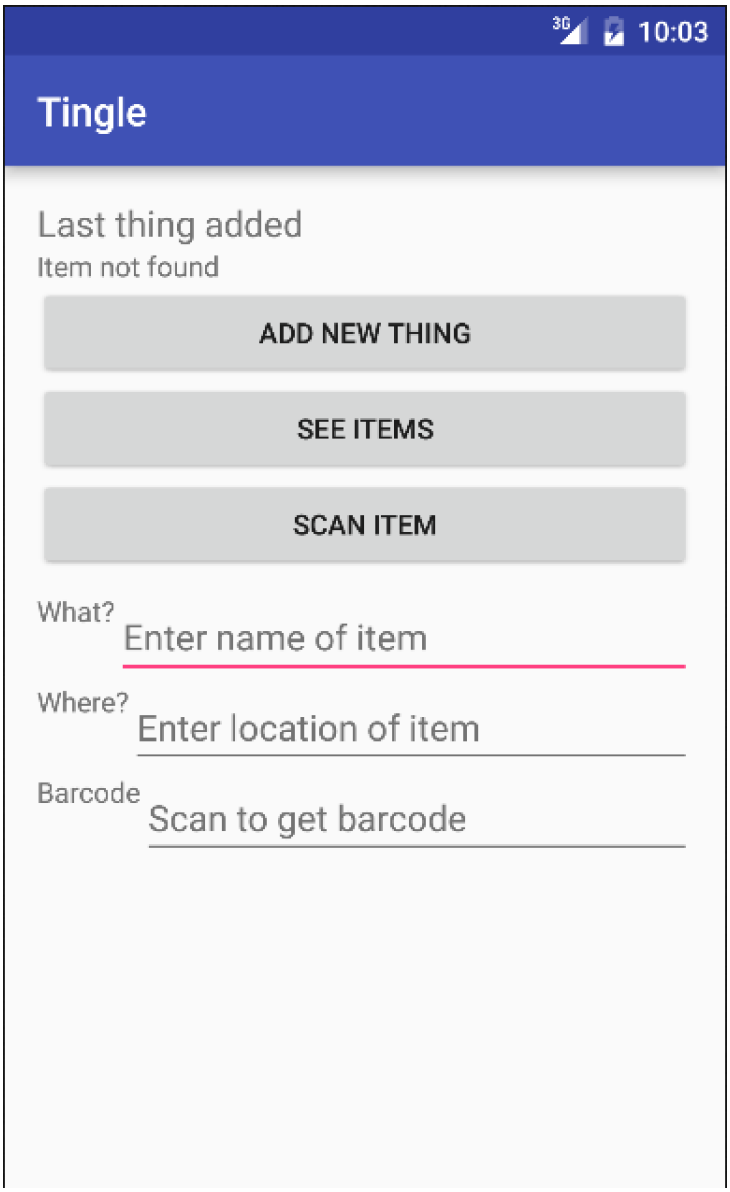

Figure 6: Main page in portrait mode

A.7 Tingle App Landscape Main Page UI

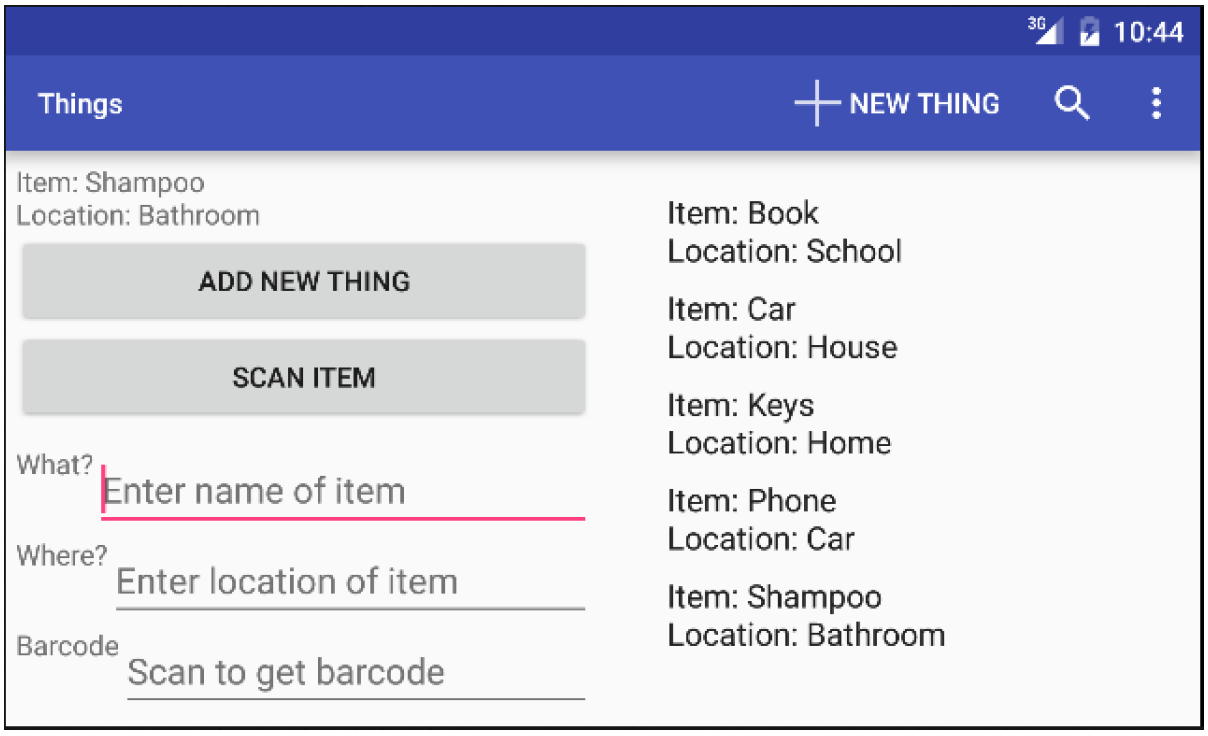

Figure 7: Main page in landscape mode

A.8 Tingle List Page UI

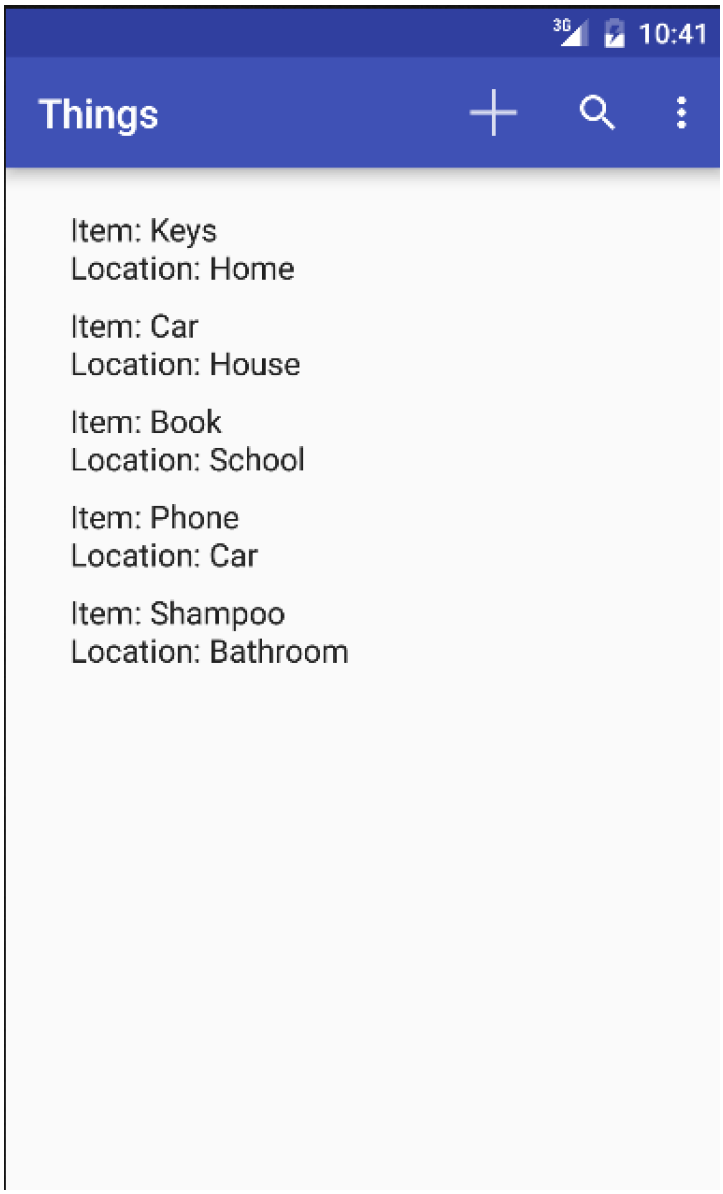

Figure 8: List page

A.9 Tingle Search and Sorting Menu Items UI

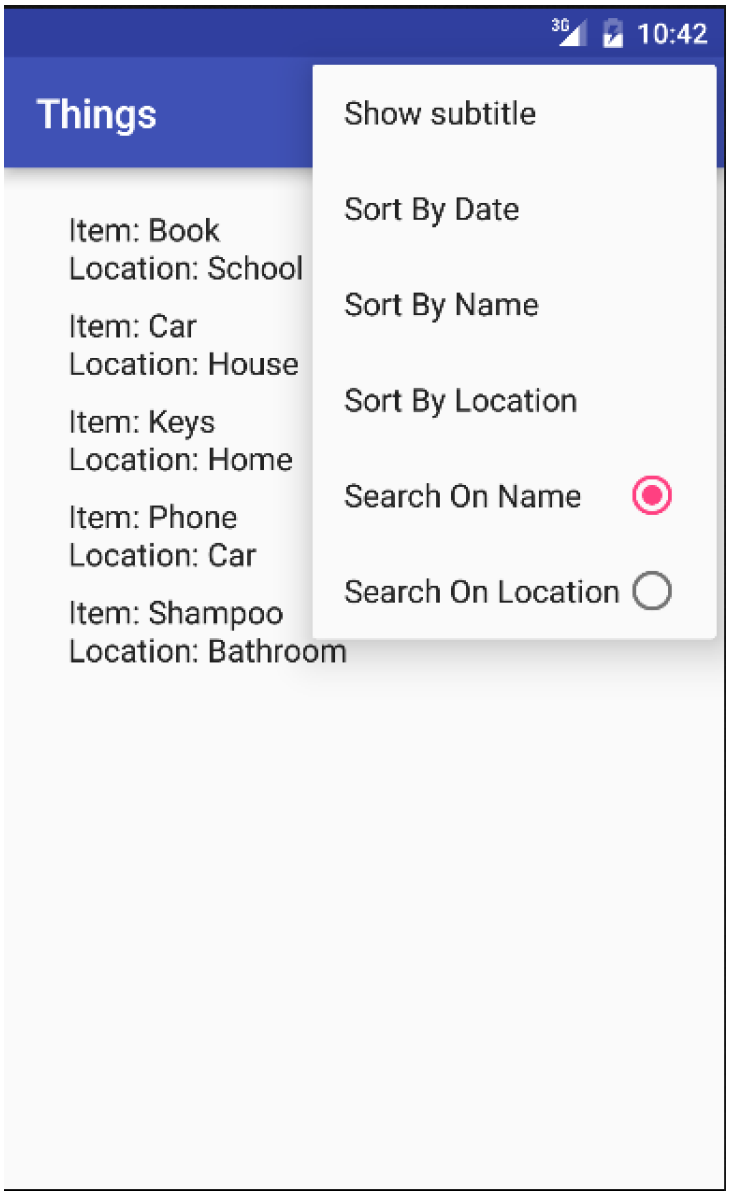

Figure 9: List page

A.10 Tingle Detailed Page UI

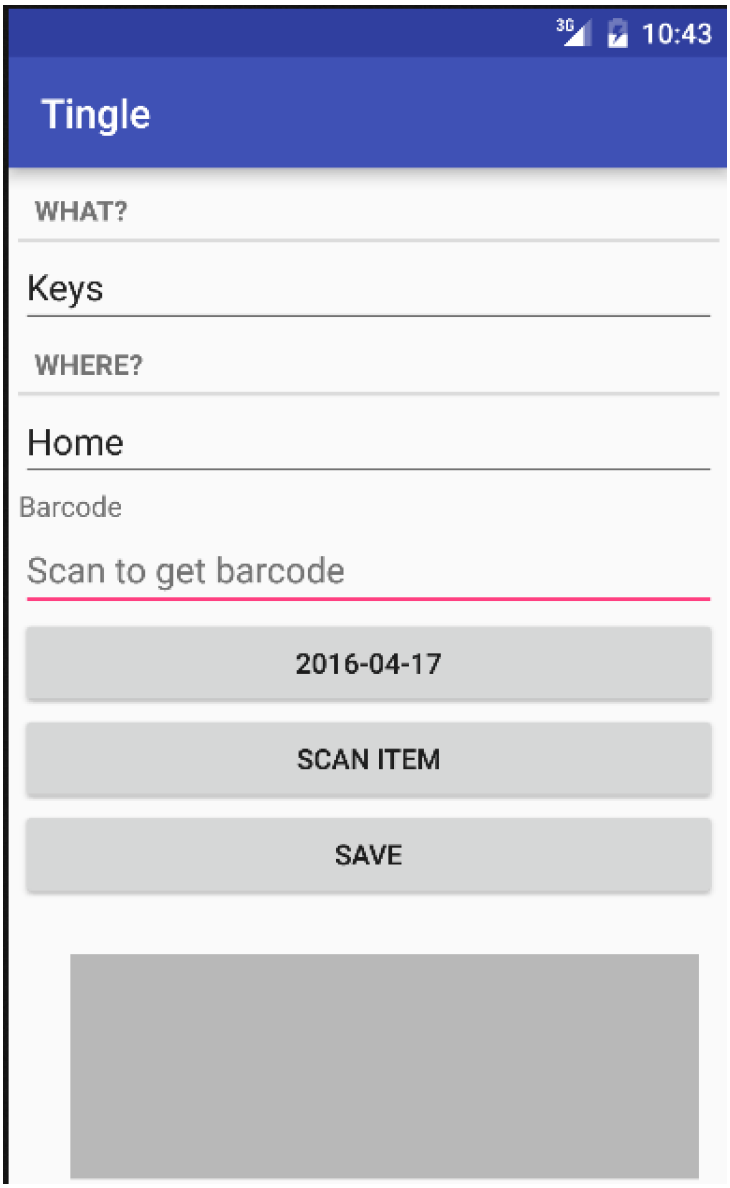

Figure 10: Detailed page