Pensive: An App to Remember

Gerben van der Lubbe (s4389026)
Lars Jellema (s4388747)
Rick Schouten (s4323750)

June 27, 2014

Contents

1 Preface

This document describes the concept, design and development of the app named "Pensive". Pensive is an app produced by the group named "The White Walkers" for the subject "Research and Development" for Radboud University Nijmegen.

The first section of this document describes the project. Here, we explain the concept of Pensive and what terminology we use to describe its features. Then we explain why we believe Pensive is something new and inventive, and what features makes it unique, followed by both the features that are functional at its current, prototype state and which features are not yet functional but considered a high priority to include in the near future. The next section contains the design. The design is divided in three components, each of which is covered separately, for which we will discuss the requirements followed by its design, including motivation. In the last section we reflect upon our experience of creating this app, including which parts we are satisfied or unsatisfied about, and what we would do different for future projects.

2 Description

2.1 Introduction

In this introduction, we will give you a description of Pensive in its current state. This is followed by the section "unique features", in which we will provide more information about why the application is inventive and in which way it provides features that other applications lack.

2.2 Features

Pensive is an application that allows the user to create lists in which users can store information for later retrieval. Each list consists of "items", each of which contains one or more "fields", each field having a certain name and type, in which the user can store something of said type. For example, an item could have a field called "title" of type "string", which allows the user to type a string which is related to the name "title". When the item is displayed, a "title" field would be displayed with the value the user entered for that field.

An example of what pensive looks like in use is shown in figure 1.

Only two standard[1] types of fields are currently available: strings, where the user can enter a text, and rating, where the user can give a rating of 1 through 5 stars. More fields were planned, but are not included in the prototype.

In each list the items have the same fields available and the same layout in which these items are presented to the user. The combination of the layout and the available fields (including name and type) is called a "template" in Pensive. Hence: each list has a single "template" assigned to it.

One special type of template is the "list template." This template allows a list to be created in which more lists can be created. This way, lists can be nested, similar to directories on a filesystem.

Furthermore, the application allows the management of such lists, such as adding, editing or deleting items.

[1] Technically, another type is available, where one can specify the template for lists, but it's not usable in regular items

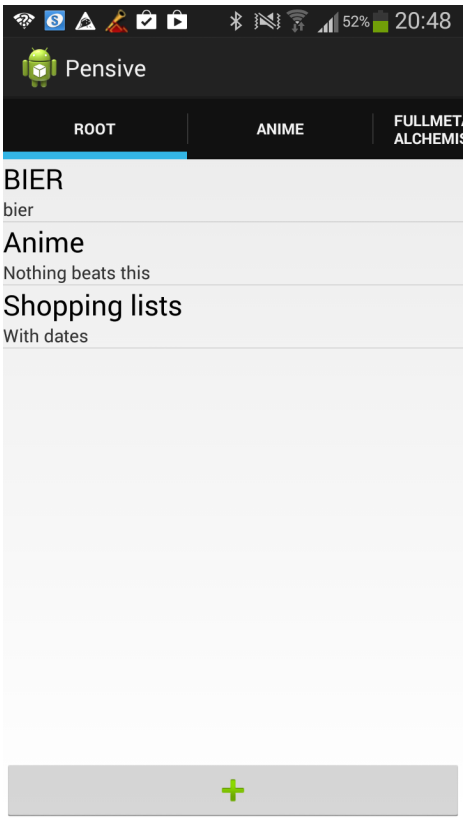

Figure 1: An example of what the interface looks like during regular use

Henceforth the term "list" shall be used for both the concept (a collection of items) and the visual representation of lists (where the items are actually displayed to the user).

2.3 Unique features

Here we shall describe the unique features as had been planned for the final version of this application. Note that due to the limited time available for development of this application, not all these features have been included in the prototype.

There are already many apps to take notes and store them, as well as apps to create grocery lists and todo lists. But what makes Pensive unique is its great flexibility: Lists can be created in all forms and shapes, not just grocery lists and todo lists, but also for example, rating lists for beers or movies, lists of bookmarks and many other lists. While creating new list types is something for more advanced users only, as it requires some knowledge of xml and the

specific markup language we use[2], these list types can be shared among all users, making its flexibility available for everyone. Furthermore, it would seem that a small number of templates would be sufficient for most of the scenario's we've considered the application to be used for.

The advantage that these custom lists provide is the fact that sorting and searching is much easier. Where in a regular list app you'd need to put all information in one text box and then search for literal strings, Pensive allows you to make a neat distinction of the data you enter. For example, if you have an item template for movies with a title, a description, a rating, a genre and a date at which you watched the movie, you can search your list specifically for certain keywords in the genre field. It will only return results that match in the genre fields, and no results that match in another field, as that's not what you were looking for.

But it gets better: Because Pensive knows that the date field holds dates, it can provide the user with more specific sorting and searching options. For example, a feature could easily be implemented to search for all movies you watched on a Thursday, or to search for movies you watched in the period of one week before your birthday until one week after. This way, if you remember you watched a really nice movie somewhere close to your birthday, but you don't remember the name, you can still find it with ease. As a bonus, you can easily see how many movies you tend to watch in two weeks.

If you feel like watching an old movie again, you can sort your list of movies by watch date in reverse, and watch one of the movies you haven't seen in a long time. Or if you feel like watching a new movie, it can be useful to combine searching and sorting: First search for movies that have no watch date (movies you haven't watched yet) and are in the romance or comedy genres because that's what you feel like watching. Then you can sort the results by rating so you can select the best movie you haven't seen yet.

This is just one example of many possible types of lists, all with their own useful searching and sorting options. All of these lists can be neatly organised in a simple directory structure and because directories are simply lists of other lists, all the useful sorting and searching options apply to these as well. We believe that this flexibility is what makes Pensive truly unique and so much more useful than the many other apps already available on the market for a large number of scenario's.

2.4 Specifications

2.4.1 Functioning properties

UC#:	UC01
Name:	Navigation
Description:	This use case describes the navigation through the lists and items of Pensive.

[2]This will be described in more detail later

Primary Actor:	The user of the application.
Trigger:	The user launches Pensive.
Basic flow:	
Step	**Action**
1	The system displays a tab named “root” and the list of items without parent.
2	The user can either select a tab, swipe left, swipe right or click on an item in the currently displayed list. If the user selects a tab sub-flow 3.1 is executed, if the user swipes left sub-flow 3.2 is executed, if the user swipes right sub-flow 3.3 is executed, if the user clicks an item that is itself a list sub-flow 3.4 is executed.
3.1.1	The system displays the list of the selected tab.
3.2.1	The system displays the list at the tab left of the current tab.
3.3.1	The system displays the list at the tab right of the current tab.
3.4.1	The system closes the tabs right of the displayed tab.
3.4.2	The system opens a new tab to the right of the active tab with as title that of the item.
3.4.3	The system displays the contents of the pressed item in the newly created tab.
4	The process continues at step 2, until the user terminates the application.
Alternative flows:	
3.2.1a	The system is already displaying the root list, so there is no tab left of the current tab. This is indicated visually and the process continues at step 2 displaying the same list.
3.3.1a	The system is already displaying the right-most list, so there is no tab right of the current tab. This is indicated visually and the process continues at step 2 displaying the same list.
Precondition:	The application has been started.
Postcondition:	The lists are in the state as specified by the user.
Remarks:	The items can be added, edited or deleted, which is excluded from this navigation based use case.

Table 1: Use case for navigation through Pensive.

The use case diagram showing the functioning properties of Pensive is shown in figure 2. The detailed use case for navigation of Pensive is shown in table 1.

Navigate means you can browse through the folders, which includes viewing lists and the list items therein. Lists and folders will have their description displayed below their name. The same goes for list items, though the difference

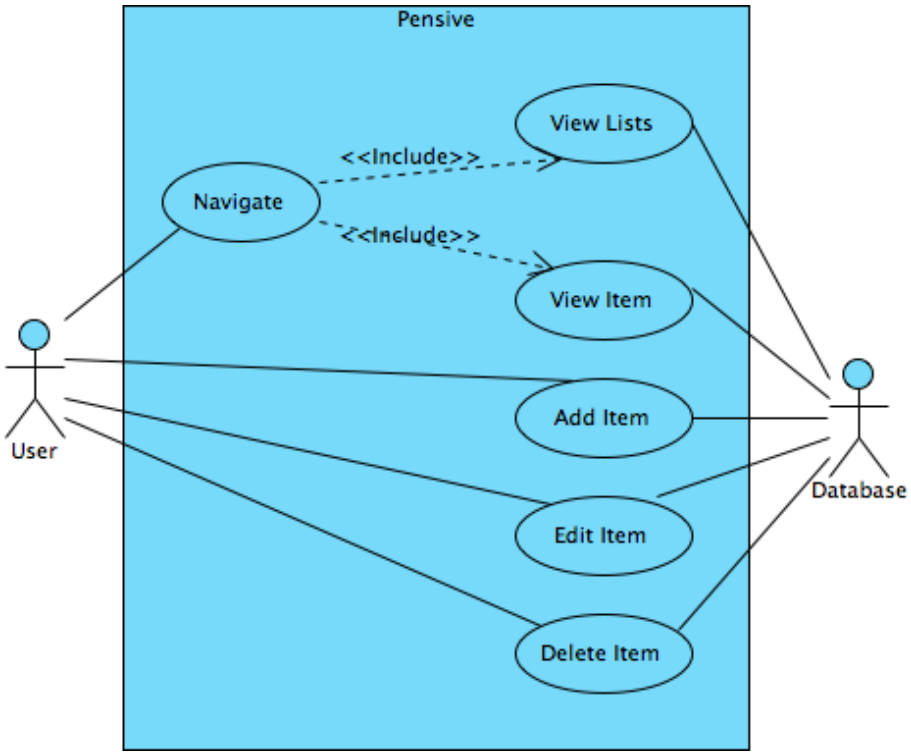

Figure 2: Use case diagram for Pensive

is that they will be modified according to the template of the list they are part of.

What navigating through folders and lists looks like is shown in figure 3. The user is in the progress of swiping between the two displayed fragments, one displaying a list, and one displaying a single item from that list.

Add Item is an option to add directories, lists or list items (all of these are items). Touching this will take you to a screen where you can edit the properties of the item, with list items this will be according to the list template. You can then choose if you want to cancel the addition, or go through with it.

What adding items looks like is shown in figure 4.

Edit Item can be accessed by touching and holding an Item and then touching "Edit" in the pop-up, or, if it is a list item, touching it. You will be taken to the same screen you used to add a new item, where you can alter the item's properties.

What holding an item, the pop-up that appears for editing and deleting looks like is shown in figure 5.

Delete Item can be accessed the same way by touching and holding an item, but this time touching "Delete" in the pop-up that follows.

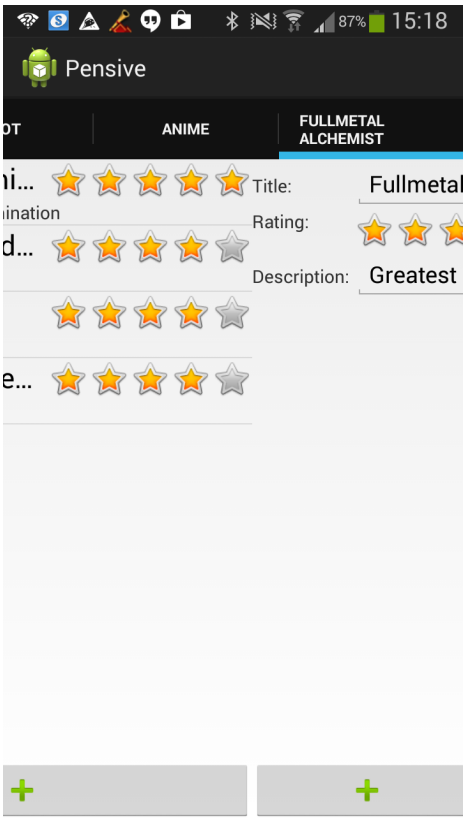

Figure 3: Example list and item navigation

2.4.2 Future additions

There is a host of features we still want to include in our app, namely the following:

- Searching for items in a list.
- Sorting the items in a list, and in search results.
- Adding new field types such as number, date, time and tags.
- Giving users the possibility to create custom templates.
- Synchronising with a server so that lists aren't lost when a device crashes/breaks, and as a first step to let users share lists.
- Sharing lists between users.

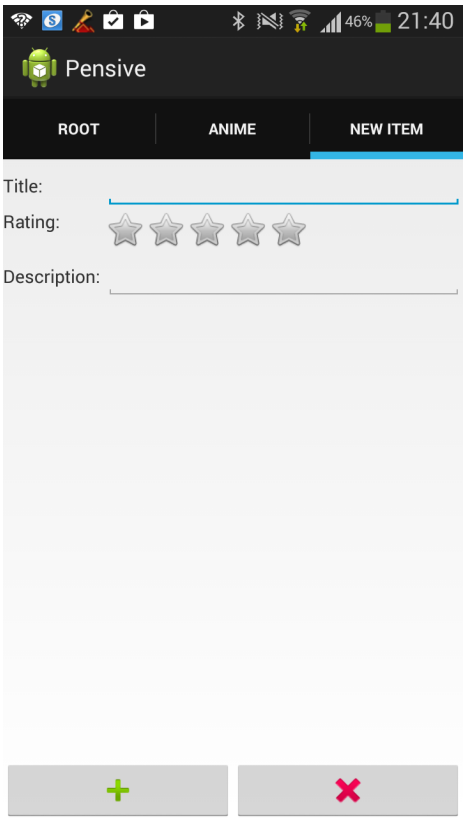

Figure 4: Example of adding an item

3 Design

Pensive can be divided into three components, the details of which will be covered in subsequent sections. These three components are as follows:

- Layout Builder: Android doesn't have support for rendering dynamic views. Any layout must be contained in the binary file, compiled along with the source of the application, and thus must be known at compile time. This would make it impossible to support custom item templates, however, which was a mandatory requirement, at least as possible later addition.

  While dynamic layouts are possible through components that render HTML, they have proven[3] to be too slow for our application, as the HTML must be parsed for every item in the list. Furthermore, the style of rendered

[3] We have conducted a test by creating a list with WebView elements with slightly complex HTML documents. When scrolling quickly the items loaded noticeably slow.

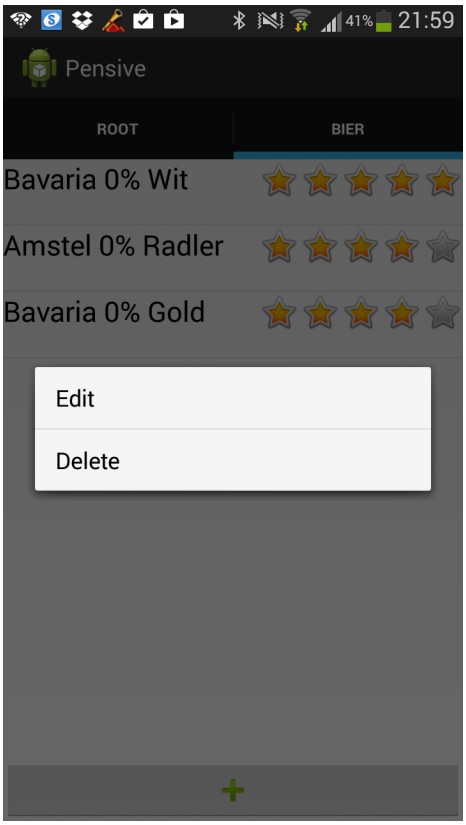

Figure 5: Example pop-up for editing and deleting items

HTML would be distinctly different from the rest of the application that is based on Android widgets.

We have overcome these problems by implementing a custom library that allows the creation of dynamic layouts efficiently.

- Item Management: Item management is the component responsible for the storage and retrieval of items, templates for these items, and lists of items in the form as defined by a certain template.
- Application Logic: The remaining logic exists to combine these two. These other components are used to retrieve the list of items to display and the layouts to display them with. This component handles the transition and navigation logic for the lists and items.

3.1 Layout Builder

3.1.1 Discussion

The layout builder is responsible for supporting dynamic views. This allows the creation of item templates that are not known at compile time, such as by the user directly.

The considered requirements for the layout builder were as follows, in order of importance:

1. Speed: The dynamic views must load reasonably fast; scrolling through a list should not be noticeably jerky or display views that are not yet fully loaded.

2. Flexibility: The layout builder must have a large degree of freedom over the resulting layout, as in that any reasonable layout requested by the caller must be constructible by the layout builder in its final form.

3. Ease of use (user): While creating custom layouts is only feasible for power users[4], these power users should be able to do so without extensive knowledge of the internal workings of the application.

4. Extendible: During and after development many additional display elements or attributes for these elements should be added. This should be possible with reasonable ease.

5. Separation of responsibility: Portions of the builder that can be separated logically should be separated. More specifically; large portions of such a system could easily be reused for different applications where dynamic views are needed. In order to do so, no code should need refactoring.

6. Portability: The code should be portable to other systems without requiring much refactoring. Android specific code should be separated from portable Java code.

7. Ease of use (developer): Developing code that uses the Layout Builder should not be overly complicated.

To allow for a reasonable speed, which the WebView could not, as noted earlier, it is important to consider in which way the rendering of HTML is different from the requirements for the Layout Builder. The issue causing the sub-optimal speeds of the WebView is the fact that it parses the HTML for each instance (so for each item in the list), while in our scenario each item in a single list will be described by identical views with merely different information displayed. The WebView has no knowledge of this fact, the exploitation of which can be used to greatly speed up the creation of views.

[4]While, technically, it is possible to create a WYSIWYG (what-you-see-is-what-you-get) editor, allowing non-power users to create layouts, the development of such is not feasible in a reasonable time.

In order to take advantage of this fact, the layout builder required two forms for the description of the layout: one in a human-interpretable language for manipulation, and one "compiled" form, essentially a cache for the layout. The translation of the first to the second form is slow, while the translation from the second form to a view is quick[5].

Here lies the required optimisation for Pensive in relation to the WebView: the latter has no possibility to support this compiled form. By implementing this cached form, we provide for a significant speed boost[6].

The described solution does not impose any restrictions on flexibility: it can still remain as flexible as HTML or as Android layouts, given enough time to implement such features. A large degree of flexibility can be achieved by supporting a markup language similar to HTML or Android's XML for layouts.

As it is not feasible to support all features of HTML or Android's XML for layouts, the user should not be confused by allowing a mere subset of one of the two. Therefor, we decided on a merger of the two, resulting in an XML-based markup language with custom elements and attributes.

In order to maintain extendibility we have designed the layout builder in such a way that the supported tags and attributes[7] can easily be added to. In agreement with the separation of responsibility the list of supported tags and attributes are maintained at a single place in the source code.

The layout builder could partially be useful for different applications requiring dynamic views, while other parts of the layout builder, such as those for displaying items, are specific to Pensive. This suggests another significant requirement for the separation of responsibility, namely the separation of this Pensive-specific part of the generic part. This would allow the non-Pensive specific part of the layout builder to be used by other applications without refactoring or changing the code, and the code could be extended in a similar method as Pensive to implement some custom features for the layout builder.

3.1.2 Design

The class diagram of the most important portion of the layout builder is shown in figure 6. The LayoutParserXml class is used to convert the XML to a class implementing the interface ILayoutDescription. A class implementing this interface is a cache of the layout, in a "compiled" form, which can quickly be translated to a view.

The LayoutParserXml also requires as a constructor argument a class implementing the interface ILayoutDocumentType. This class allows a method to specify the attributes and elements that are supported for this type of document, and the creation of the proper class instance of the document, extending

[5]This merely requires the traversal of all elements and creation of the view, which are both required for the regular method of Android's internal generation of views, which has proven to be sufficiently fast by many applications

[6]The scrolling was immediately fluid and items loaded immediately

[7]The different types of components that can be displayed, and information about specific extra attributes manipulating the details for this component. The same concepts are used in both HTML and Android's XML for layouts.

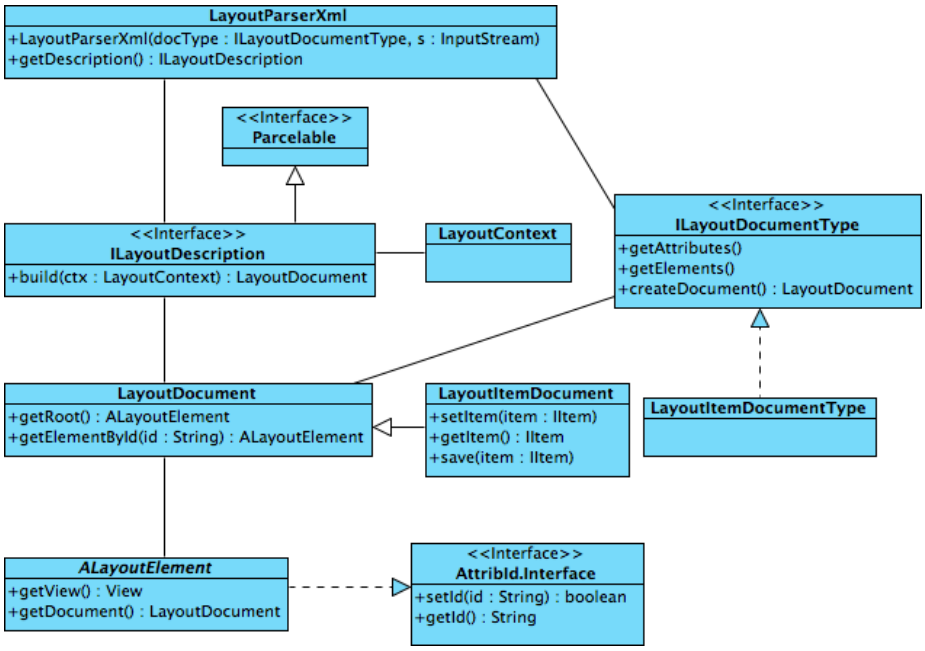

Figure 6: Class diagram of the Layout Builder

the LayoutDocument class. This allows the prior discussed distinction between the features available in the layout builder portion that is portable and the portion that is Pensive specific. The shown class implementing this interface, LayoutItemDocumentType, is the document type that is specific to Pensive and creates a LayoutItemDocument, which allows an item to be written and read to and from the layout. There exists another implementation of the ILayoutDocumentType class that only allows the generic features of the layout builder, though this class is omitted from the diagram.

Note that this ILayoutDescription need not come from the LayoutParserXml. It is possible to implement different representations for a layout by building a different class implementing this interface. This isolates the responsibility of the translation inside the LayoutParserXml, and the rest of the layout builder works independent of this format.

Also note that the ILayoutDescription interface extends the Parcelable interface. The reason for this is that it must be possible to pass these classes to activities or fragments, which require the class to be either serialisable or parcelable. Parcelable classes are quicker, though this does conflict somewhat with the portability aspect, as Parcelable classes are Android specific.

The LayoutContext is simply the context in Android, and the class exists only as an isolation of the responsibility for this information, in order to make the layout builder easier to port to other systems.

A class implementing ILayoutDescription creates the proper LayoutDocu-

ment (sub)class instance. The document can be considered to be similar to an HTML document: it can be used to retrieve and manipulate elements of the DOM-tree forming this document. The LayoutDocument is not, directly, a view: it merely contains the view, alongside with more information on the elements making up the document and view.

Finally, the diagram shows the method of adding supported attributes for specific elements, by simply implementing the interface for the attribute, allowing additional attributes to be added to elements without significant effort for the programmer. Every element supports an "id" attribute, as shown, in order to identify and quickly retrieve specific elements of the document.

Additional classes are omitted, as they are only used internally by the layout builder.

This diagram is in agreement with all of the design considerations, although the portability is still somewhat limited. As noted earlier, the extending of Parcelable limits the portability to non-Android systems slightly. Also, the "View" class is returned directly by the ALayoutElement class, even though different systems may have different names for a similar class.

3.2 Item Management

3.2.1 Discussion

The item management is the component responsible for the storage and retrieval of items, templates and lists. While the current version of Pensive only supports displaying full lists in order of creation, the original scope of the project also contained the searching through and sorting of lists. While these features have not yet been implemented, the design has been based on the support of these features such that it could easily be implemented in the future.

The considered requirements for the item management were as follows, in order of importance:

1. Ease of use: Where limitations on items, lists or templates make the application in its final form easier to use for the average user, these limitations should be applied.

2. Flexibility: The items should be flexible in that as few as possible limitations for items should be applied on the user, unless it affects the ease of use for the average user of Pensive as discussed in the previous point.

3. Extendibility for additional requirements: Not all wanted features have been included in the prototype version, yet we considered it important to design the system in such a way that these features can be implemented later.

4. Extendibility for additional features: Features not specified in the requirements should not be overly complicated to include.

To ensure that Pensive is relatively easy to use, we placed a significant limitation on the storage of lists, namely that every item in a list must be of the same template. The reason for this is that it's both easier to glance over a list if each item shares the same format and that it's more intuitive to preform a search action through lists where each item contains the same fields[8]. For exactly the same reasons it was also decided to limit the searching for items through only a single list, and not recursively through a number of lists.

In order to remain extendible within the scope of the future requirements we considered it important to maintain a method of not only having groups of items in a single list, but also (possibly ordered) subsets of items of a single list, in the form of search results. Note that a list is always a subset of itself, hence this method is usable for retrieving both full lists and items from search results.

3.2.2 Design

The class diagram of the most important portion of the item management is shown in figure 7. Only the most important methods are listed. The remaining classes are merely implementations of the interfaces that write and retrieve information to and from the database.

The IObjectSelection and IObjectSource are both interfaces for classes that store a (possibly ordered) subset of a list. The latter, IObjectSource, allows the retrieval of the number of items in the list, retrieval of an item at a certain position in the subset, and the retrieval of information about the template that is used for each item in the list (as discussed earlier, there can only be one per subset of objects, as they must exist in the same list as discussed earlier).

For the IObjectSource realisation to be able to retrieve these items a database connection needs to be available. To open a database connection an Android Context class is required, which can not be serialised. As such, an instance of such class may (and probably will) not be able to be stored to for example shared preferences or bundles. This means that these classes can not be passed to activities or fragments easily, something that is required for Pensive to indicate to a fragment which items should be displayed in a list.

In order to solve this problem the IObjectSelection interface is used. The class stores sufficient information about a filter, such as part of a query, for the IObjectSource to use to retrieve the items passing this filter. The information for the IObjectSelection should (and most likely is trivially) parcelable and as such can be easily passed to activities or fragments or stored to for example the shared preferences.

The class implementing the IObjectSelection interface requires but one method, which takes as argument a context which can be used to create an IObjectSource instance, for example to establish a database connection. As the IObjectSource will likely hold resources it implements Android's Closeable interface which must

[8]For example: if one item in a list has a "description" field and another a "notes" field, the user may be confused which of the two fields he used for the specific string he wishes to search for.

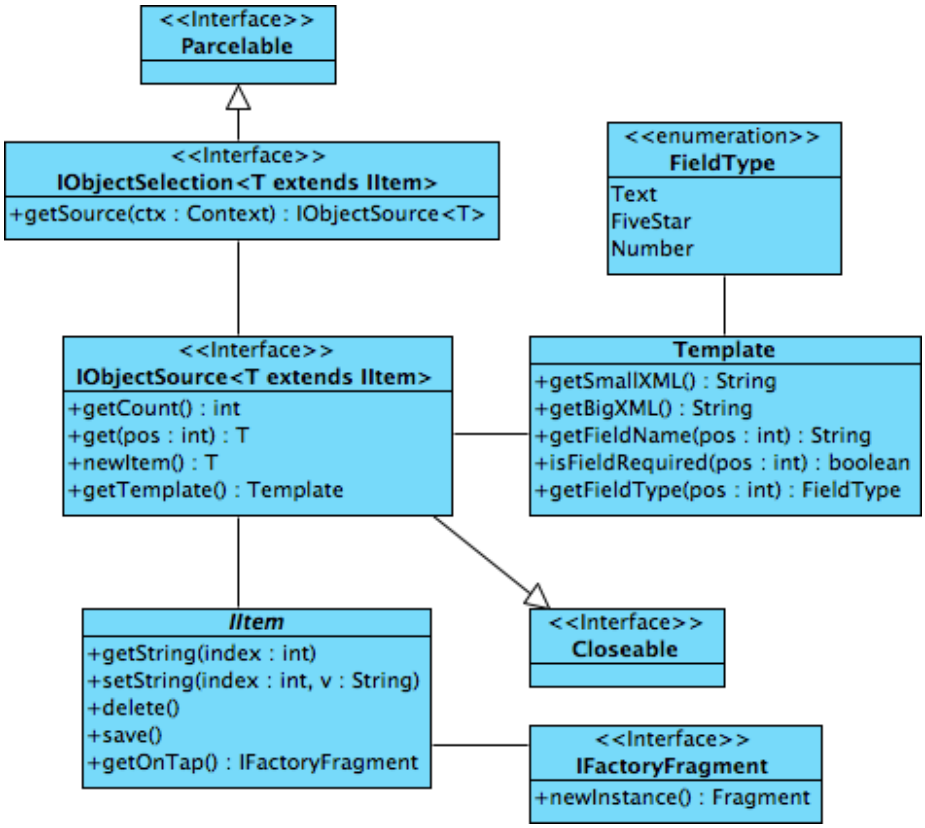

Figure 7: Class diagram of the Item Manager

be implemented to release the resources when the IObjectSource is no longer required.

From the IObjectSource's implementation an IItem or a subclass thereof can be retrieved. Not limiting IItems only allows special types of items to be implemented with relative ease, though in the prototype only a single class implementing IItem is used.

The items consist out of a number of values of a specific type. To allow easy retrieval thereof, each item contains these values as fields with numeric indices starting at zero. Using this index, values can be read or written (as shown for string values). The template stores information about this index, such as the name and type (as seen in the FieldType enumeration) of this field and whether the field is required or optional when adding a new item.

To enhance the flexibility a method has been added that decides on what fragment should be opened when the user clicks on this item in the list. It returns an instance of the class IFactoryFragment that creates a new fragment to display when needed. This way the actions that occur when clicking an item

can be easily extended to include additional features, such as opening a fragment of another application. This IFactoryFragment is also Parcelable, so that it can even be stored in the database.

Furthermore, the Template contains two methods to retrieve the XML indicating the layout of the small view (used when the item is displayed in a list) and a big view (used when the item only is displayed).

3.3 Application Logic

3.3.1 Discussion

The part that combines the item management and the layout builder into a single application is the "application logic." The design of this component came natural as a result of the design of the other two components and the choice of methods of navigation.

For navigation we decided on the "action bar tabs" with "ViewPager". The former allows tabs to be displayed at the top baring a title for every open fragment, while the latter allows the users to swipe between these tabs. This way, the user can open lists within lists, and easily navigate between the parent and child lists by swiping or clicking the titles of the wanted item or list on top.

3.3.2 Design

The class diagram of the most important portion of the application logic is shown in figure 8.

The PagerAdapterNavigate class implements the PagerAdapter to allow swiping between the fragments. As the state (such as the filled in details in an item or the position in a menu) needs to be stored when the user navigates to another fragment temporarily we used the FragmentStatePagerAdapter, though due to a bug[9] we were forced to create a copy with some slight modifications, and dubbed this class "FragmentStatePagerAdapterHacked."

The IFactoryFragments, the interface implemented by classes that indicate what fragment should be displayed when an item is clicked on, can be added and removed from the PagerAdapterNavigate, displaying or removing a list from the user's view.

Currently there exist two classes implementing the IFactoryFragments: a factory for a fragment that displays a given list of items (FactoryFragmentItemList) and a factory for a fragment that displays only a single item (FactoryFragmentItemSingle). The latter has an additional boolean value to indicate whether the item is a newly to be created item: the normal edit and add view are usually identical, with the exception of the data being stored in a new or already existing item.

[9]For more information about this bug, see: http://speakman.net.nz/blog/2014/02/20/a-bug-in-and-a-fix-for-the-way-fragmentstatepageradapter-handles-fragment-restoration/

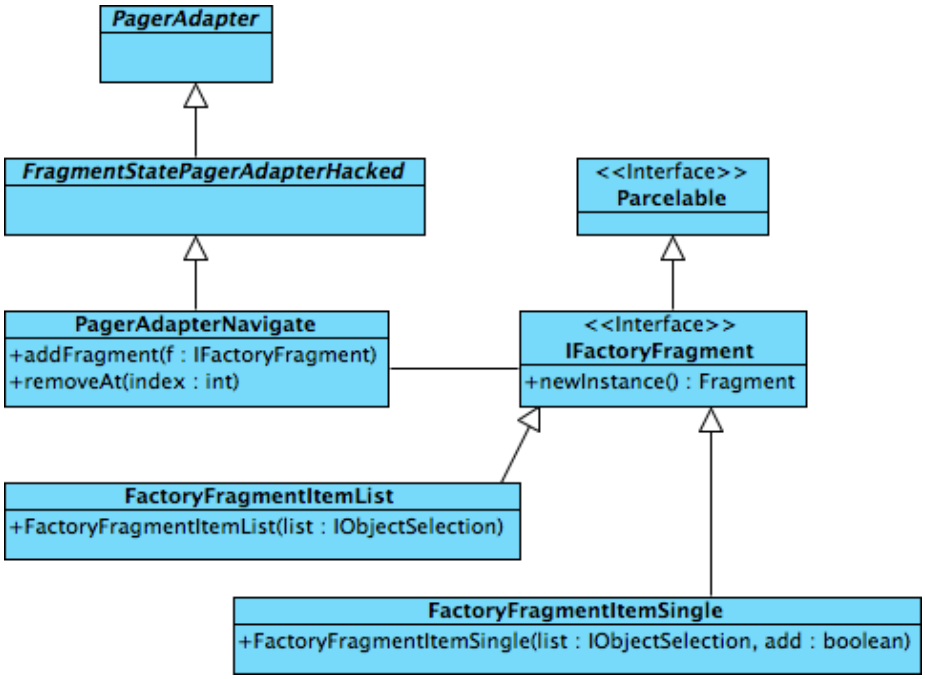

Figure 8: Class diagram of the Application Logic

4 Reflection

The first app we've written together was a simple game called The Setting Sun. Despite the fact that all of us were new to the Android platform, we managed to finish this app within a single week and deliver a polished looking app.

The programming for both apps went pretty well. We could cooperate easily as we split each app in parts connected through interfaces, allowing each of us to work on their part separately. As we needed more features, we expanded the interfaces and wrote the code on both sides to match.

As we progressed with writing our app however, we did start noticing we needed to refactor a lot of code pretty often. This made us conclude we should have put some work into the design of the code before beginning with programming. We had intended to do this too, but it was pretty hard to do as none of us really what capabilities Android has.

Right before the deadline, we hit a bug in android, as described earlier. It took a little effort but we managed to solve it by copying the android source code and modifying it.

In the end, we couldn't finish everything we wanted for the app, but we did finish an important part: An easily extendable system for the item templates and a good backend for storing all the data. Together this makes a strong base from which we can continue if we want to.

To finish this app, we'll still need to add a modular sorting and searching system that allows custom option depending on the field type. Additionally, we could keep extending the app by adding more and more useful field types.

Working on the various documents and presentations went pretty well. We just split them up in many parts and then had everyone working on separate parts in the same document. With all three of us working at the same time, these documents were usually done pretty quickly. We worked together on the presentations in a similar manner, we created the slides together, and then had the presentator do one or two rehearsals for the rest of us.

All in all, our cooperation was a success, and we're happy with the result of this project, even if it's not entirely finished. Next time however, we'll make sure to do a little more planning and design up ahead.