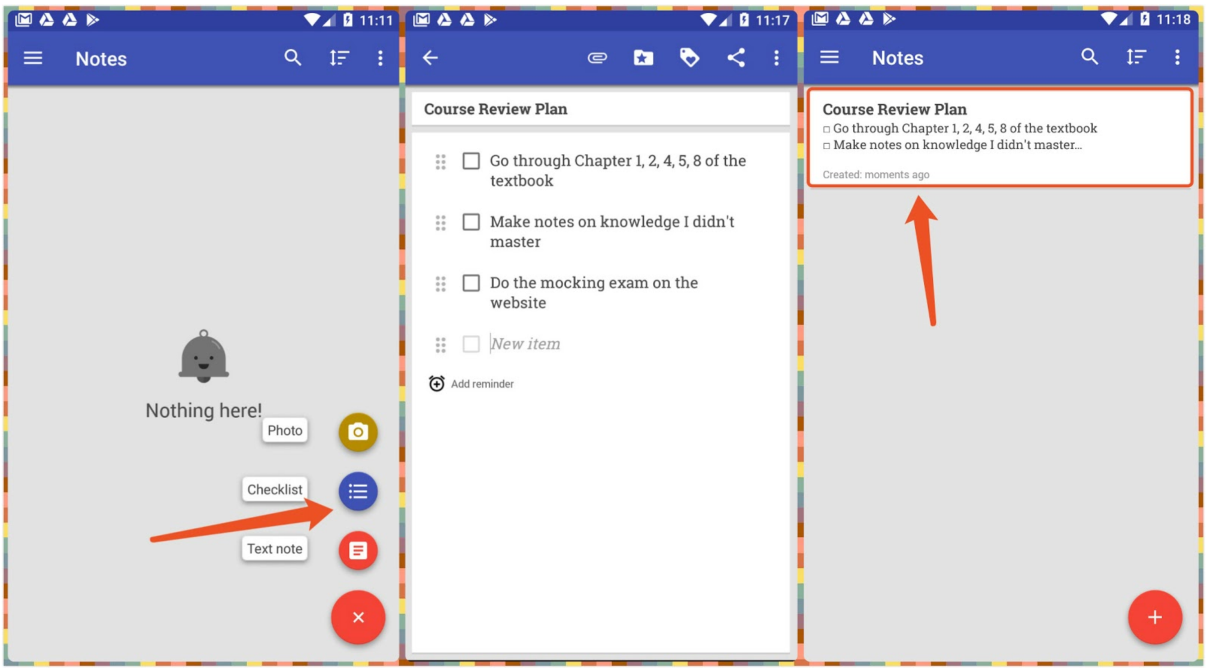
Essential features in Omni-Notes

Dongxin Xiang, Guowei Li, Jing Chen

1 Creating Notes

1.1 Introduction

Omni-Notes aims to help its users to manage their lives with various kinds of notes. It's without doubt that creating notes is an indispensable feature for this app. Omni-Notes supports three types of notes - notes with pictures, checklist notes and text notes. When creating a note, users first click "+" on the main interface and choose what type of note they want to create. Then they go into an editing interface, editing the new note with text (or pictures possibly). A non-empty note will be saved after the back button is pressed.



Course Review Plan

- [ ] Go through Chapter 1, 2, 4, 5, 8 of the textbook
- [ ] Make notes on knowledge I didn't master
- [ ] Do the mocking exam on the website

Course Review Plan
□ Go through Chapter 1, 2, 4, 5, 8 of the textbook
□ Make notes on knowledge I didn't master...

Figure 1-1-1 A user case of omni-notes

1.2 Feature Location

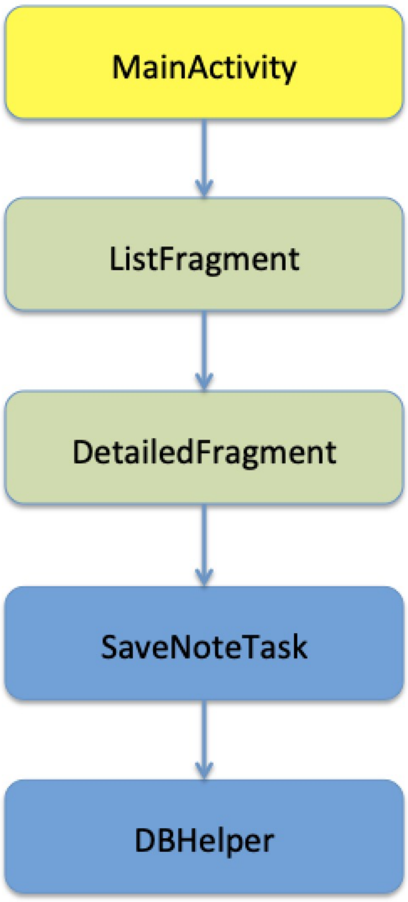
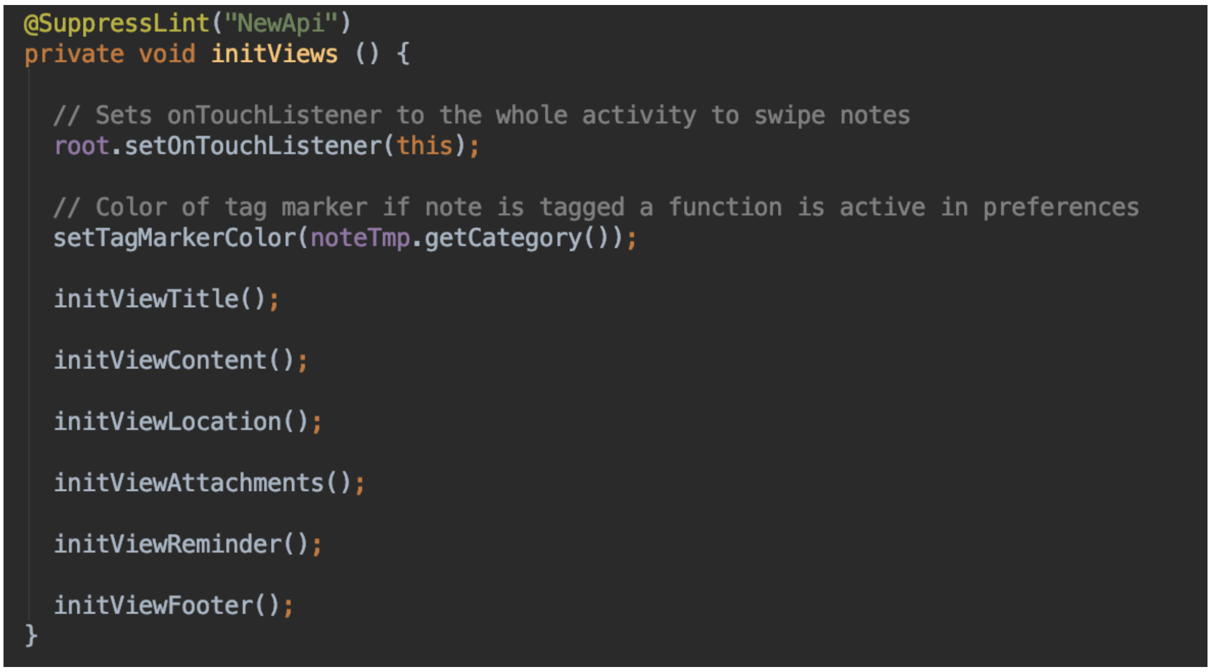


Figure 1-1-2 Relationship among key classes for creating notes

We found the code of creating a note view in *DetailedFragment.java* file. For the whole feature, *MainActivity* first sets the view with *ListFragment* to show all notes that are already created. When creating a note, the editing interface is provided by *DetailedFragment.* When a note is to be saved, SaveNoteTask will be called and this class uses another class DBHelper to save new notes.

1.3 Structural Analysis

In *DetailedFragment.java,* **initView()** (Figure 1-3-1) is called during the new note initialization. This method initializes everything in the note view and **initViewTitle()** (Figure 1-3-2, Figure 1-3-3) and **initViewContent()** (Figure 1-3-4, Figure 1-3-5) are two key methods it calls for editing the new note.



```java
@SuppressLint("NewApi")
private void initViews () {

  // Sets onTouchListener to the whole activity to swipe notes
  root.setOnTouchListener(this);

  // Color of tag marker if note is tagged a function is active in preferences
  setTagMarkerColor(noteTmp.getCategory());

  initViewTitle();

  initViewContent();

  initViewLocation();

  initViewAttachments();

  initViewReminder();

  initViewFooter();
}
```

Figure 1-3-1 Code snippet of initViews()

1.3.1 initViewTitle()

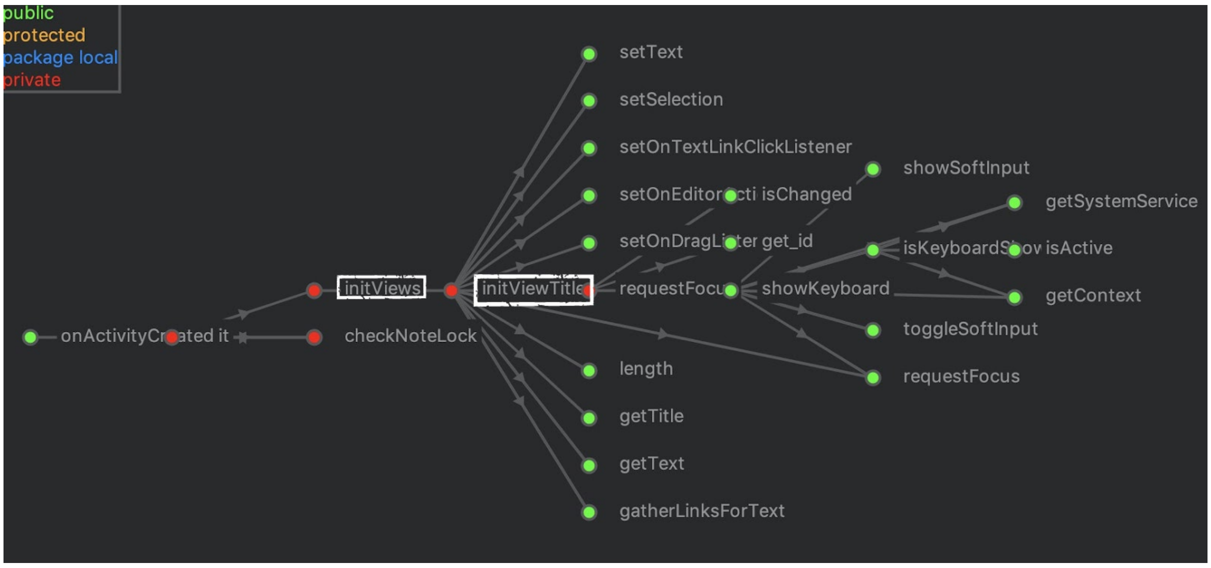
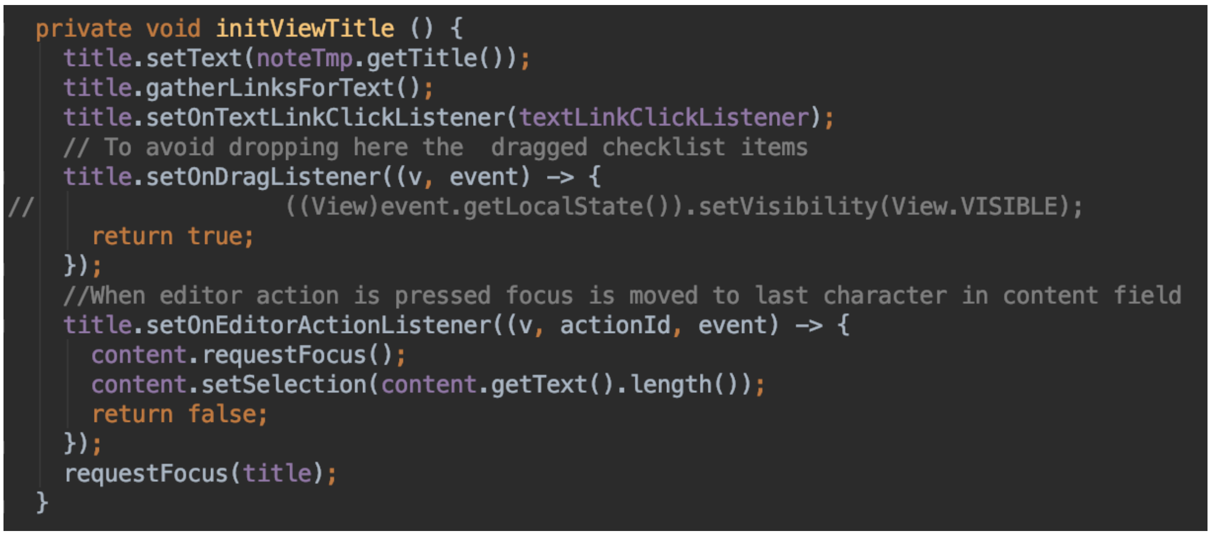


Figure 1-3-2 Call graph for initViewTitle()

initViewTitle() sets the EditText **title** with the string returned from noteTmp.getTitle(). In the case of creating a note, noteTmp is an empty note and an empty title string is returned. It allows users to simply edit the title of the note. From the call graph we can see **initViewTitle()** calls a lot of other methods. Among them, **setText, getTitle()** and **getText()** are basic and essential for creating a note. (Figure 1-3-2, Figure 1-3-3)



```
  private void initViewTitle () {
    title.setText(noteTmp.getTitle());
    title.gatherLinksForText();
    title.setOnTextLinkClickListener(textLinkClickListener);
    // To avoid dropping here the  dragged checklist items
    title.setOnDragListener((v, event) -> {
//                  ((View)event.getLocalState()).setVisibility(View.VISIBLE);
      return true;
    });
    //When editor action is pressed focus is moved to last character in content field
    title.setOnEditorActionListener((v, actionId, event) -> {
      content.requestFocus();
      content.setSelection(content.getText().length());
      return false;
    });
    requestFocus(title);
  }
```

Figure 1-3-3 Code snippet of initViewTitle()

1.3.2 initViewContent()

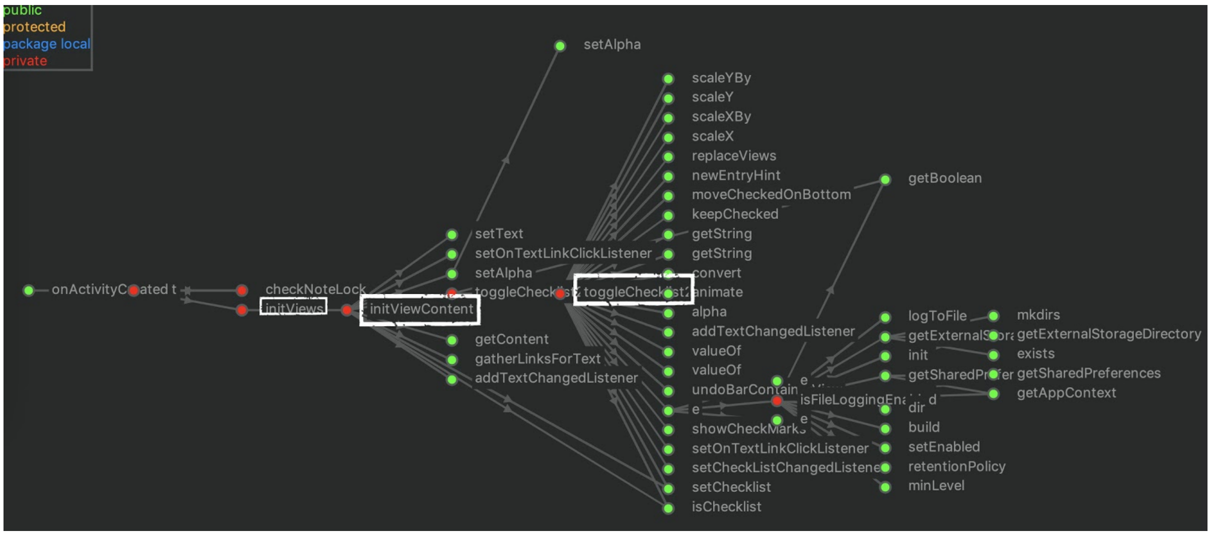
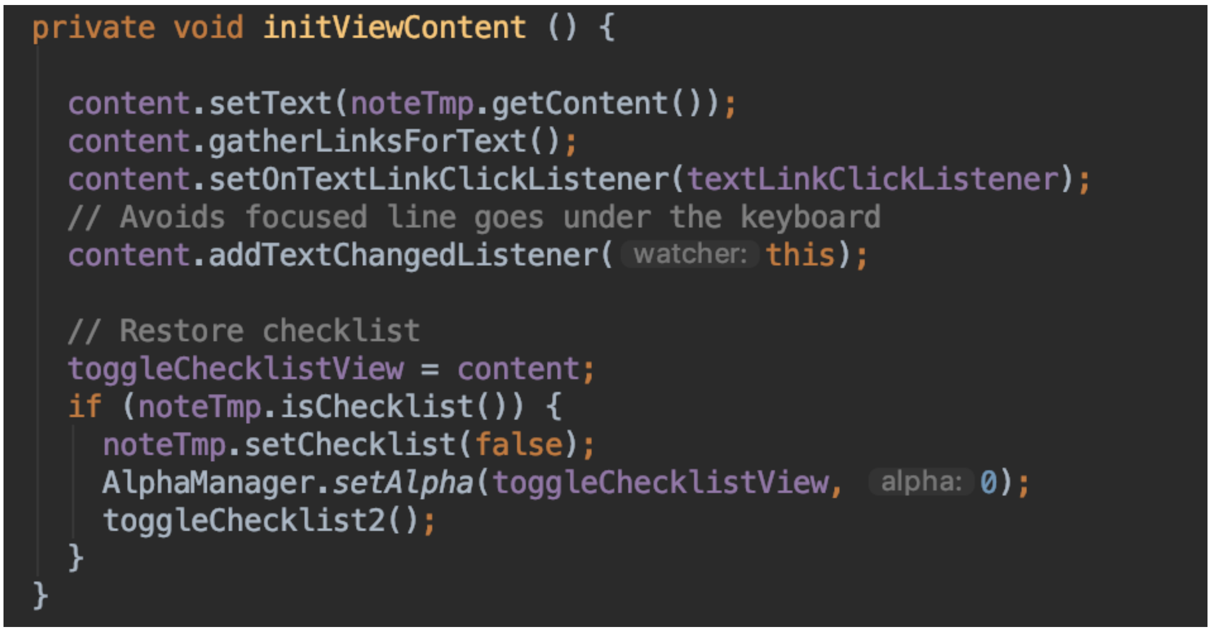


Figure 1-3-4 Call graph for initViewContent()

initViewContent() sets the EditText **content** with the string returned from noteTmp.getContent(). The same situation applies to it - noteTmp is an empty note. It allows users to edit the content of the note and mark an item as completed by clicking the checkbox. The method can separate the special kind of notes - checklist notes from others and calls **toggleCheckList2()** to deal with the boolean state of each item in a note. If it's a new checklist note, it would apply the default situation - one item with no content and an unchecked state checkbox. (Figure 1-3-4, Figure 1-3-5)



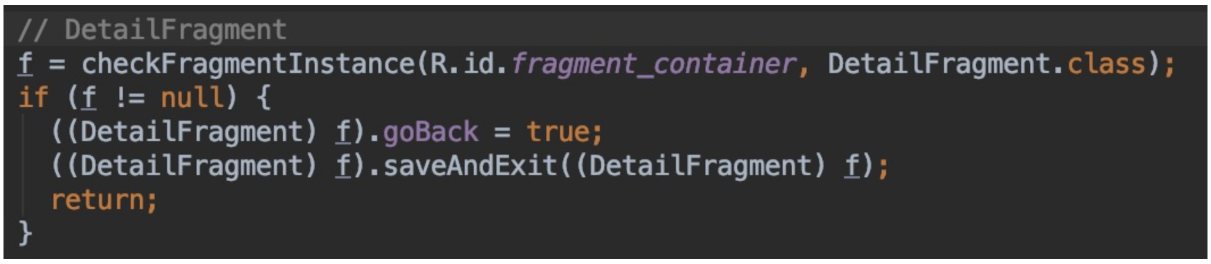
```java
private void initViewContent () {

  content.setText(noteTmp.getContent());
  content.gatherLinksForText();
  content.setOnTextLinkClickListener(textLinkClickListener);
  // Avoids focused line goes under the keyboard
  content.addTextChangedListener( watcher: this);

  // Restore checklist
  toggleChecklistView = content;
  if (noteTmp.isChecklist()) {
    noteTmp.setChecklist(false);
    AlphaManager.setAlpha(toggleChecklistView,  alpha: 0);
    toggleChecklist2();
  }
}
```

Figure 1-3-5 Code snippet of initViewContent()

1.3.3 saveAndExit() & saveNote()

As for saving a new note, by using the app we knew that a non-empty note would be saved once the back button is pressed. Both **saveAndExit()** and **saveNote()** in *DetailFragment.java* are responsible for saving notes.

saveAndExit() is called by **onBackPressed()** in *MainActivity.java*. When the DetailedFragment is loaded and users press the back button (onBackPressed() is invoked), **saveAndExit()** will be called.



```java
// DetailFragment
f = checkFragmentInstance(R.id.fragment_container, DetailFragment.class);
if (f != null) {
  ((DetailFragment) f).goBack = true;
  ((DetailFragment) f).saveAndExit((DetailFragment) f);
  return;
}
```

Figure 1-3-6 Where the DetailFragment gets called

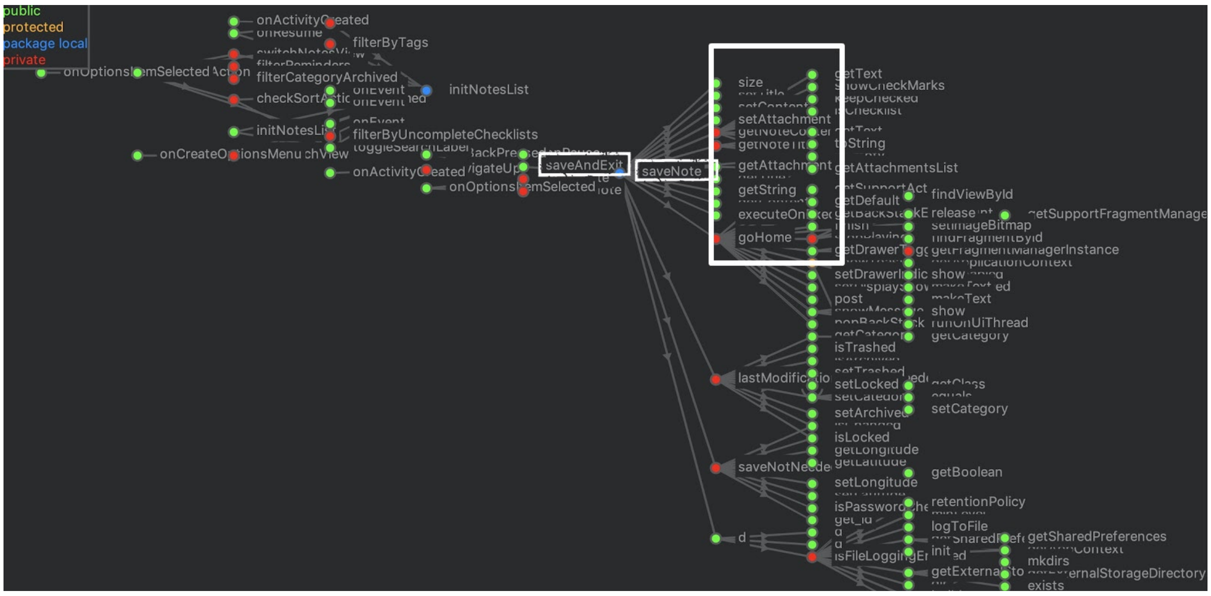


Figure 1-3-7 Call graph for saveNote()

saveAndExit() saves a non-empty note and shows a message “Note updated” at the top of the screen when users return to the main user interface. It is called the method **saveNote()** for a successful save event. **saveNote()** sets noteTmp with title and content that users edited. To get the title and the content (including the status of every item) from corresponding EditTexts and CheckBoxes, **getNoteTitle()** and **getNoteContent()** get called correspondingly. Then **saveNote()** filter all empty notes. If a note is empty, it shows a message “Can’t save an empty note”. If not, it initializes a *SaveNoteTask* (object) to really save the note.

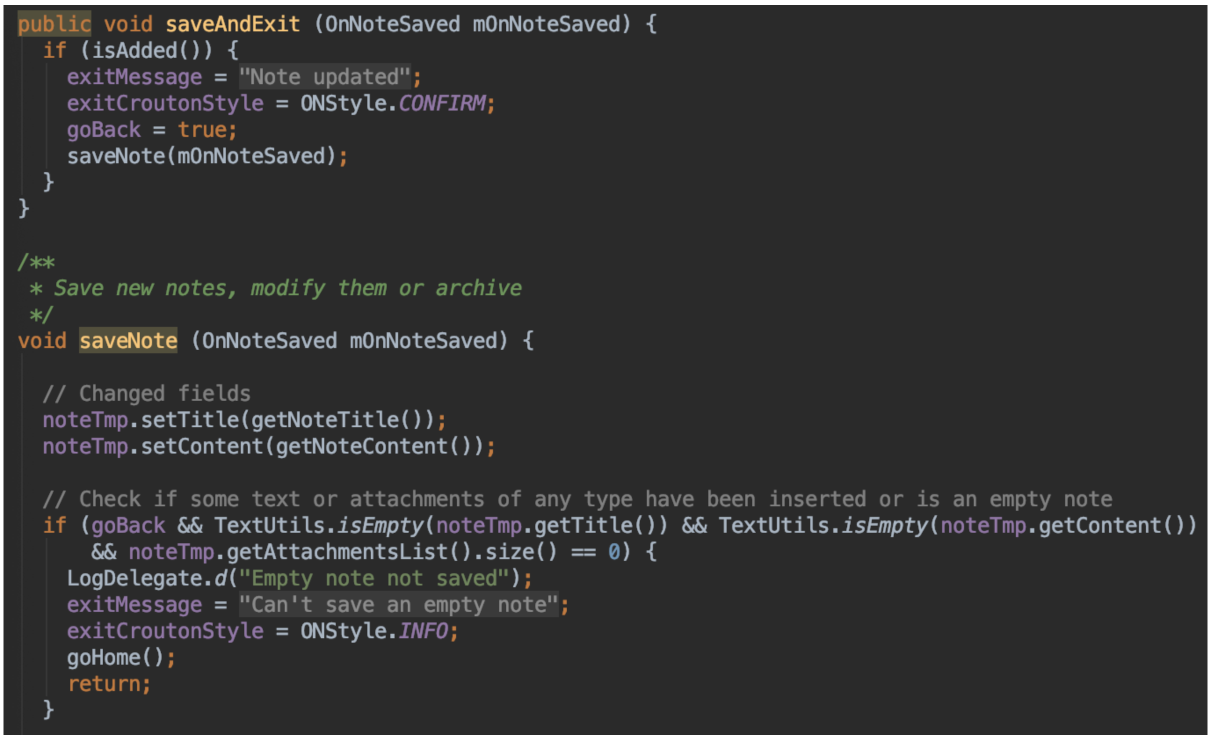
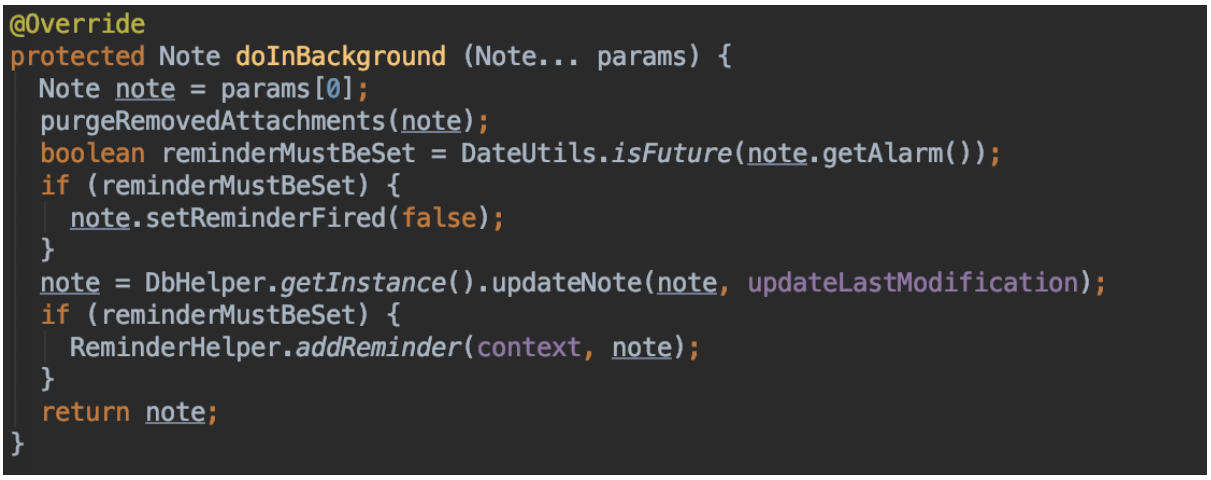

```java
public void saveAndExit (OnNoteSaved mOnNoteSaved) {
  if (isAdded()) {
    exitMessage = "Note updated";
    exitCroutonStyle = ONStyle.CONFIRM;
    goBack = true;
    saveNote(mOnNoteSaved);
  }
}

/**
 * Save new notes, modify them or archive
 */
void saveNote (OnNoteSaved mOnNoteSaved) {

  // Changed fields
  noteTmp.setTitle(getNoteTitle());
  noteTmp.setContent(getNoteContent());

  // Check if some text or attachments of any type have been inserted or is an empty note
  if (goBack && TextUtils.isEmpty(noteTmp.getTitle()) && TextUtils.isEmpty(noteTmp.getContent())
      && noteTmp.getAttachmentsList().size() == 0) {
    LogDelegate.d("Empty note not saved");
    exitMessage = "Can't save an empty note";
    exitCroutonStyle = ONStyle.INFO;
    goHome();
    return;
  }
```

Figure 1-3-8 Code snippet of saveNote() and saveAndExit()

SaveNoteTask extends *AsyncTask*. It deals with note-saving tasks in the background. For this feature, we only focus on the title part and the content part which are highly relevant to it. It writes the new note to the database in **doInBackGround()** using a *DBHelper*. (Figure 1-3-9)



```
@Override
protected Note doInBackground (Note... params) {
  Note note = params[0];
  purgeRemovedAttachments(note);
  boolean reminderMustBeSet = DateUtils.isFuture(note.getAlarm());
  if (reminderMustBeSet) {
    note.setReminderFired(false);
  }
  note = DbHelper.getInstance().updateNote(note, updateLastModification);
  if (reminderMustBeSet) {
    ReminderHelper.addReminder(context, note);
  }
  return note;
}
```

Figure 1-3-9 Code snippet of doInBackGround()

1.4 Behavioral Analysis

To finish its task, **initViewTitle()** and **initViewContent()** would interact with other components/classes clearly shown above in the sequence diagram. The calling sequence of all the methods highly related to creating notes we have mentioned above are highlighted in the sequence diagrams shown in Figure 1-4-1 and Figure 1-4-2.

1.4.1 initViewTitle()

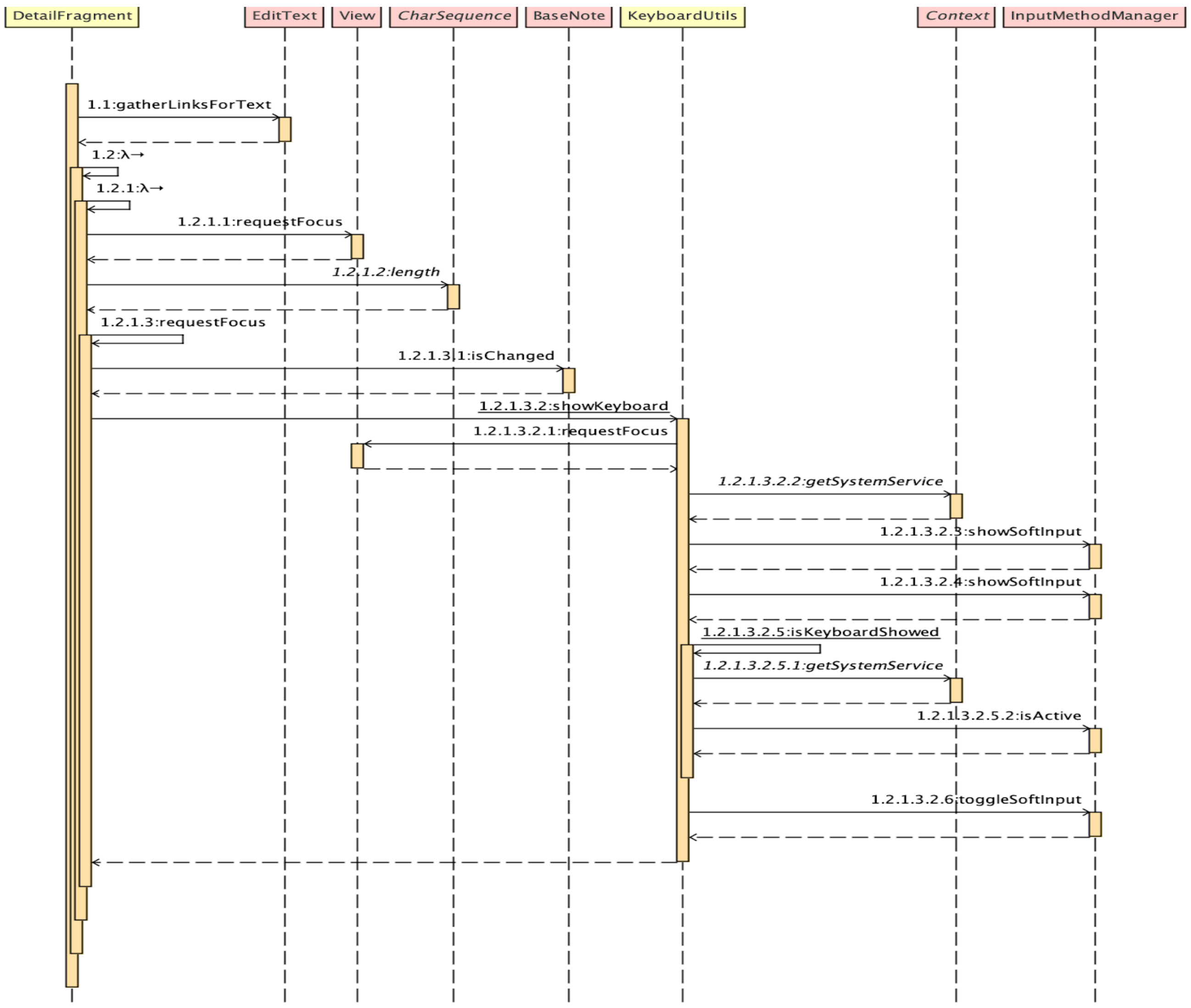


Figure 1-4-1 Sequence Diagram starting with initViewTitle()

1.4.2 initViewContent()

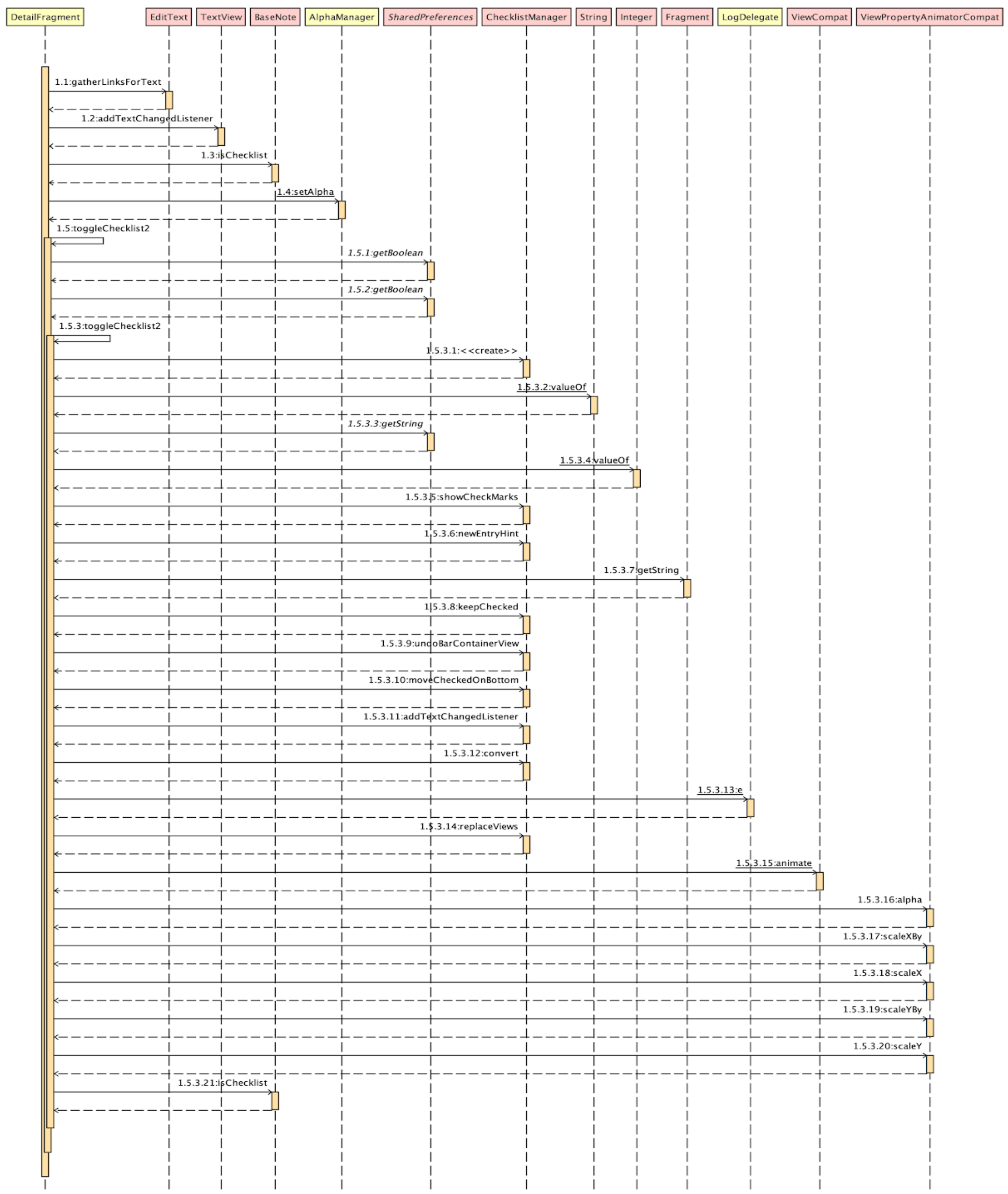


Figure 1-4-2 Sequence diagram starting with initViewContent()

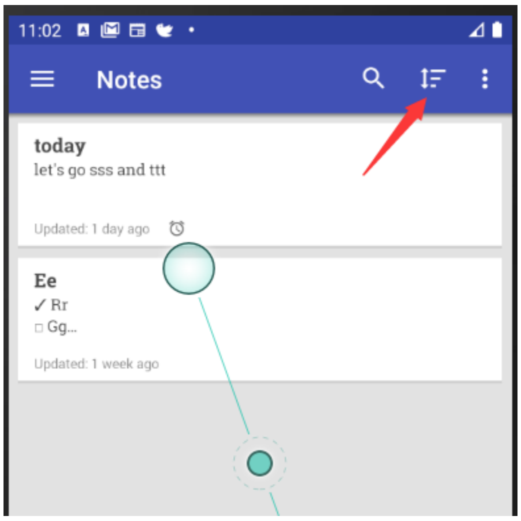
2 Sorting

2.1 Introduction

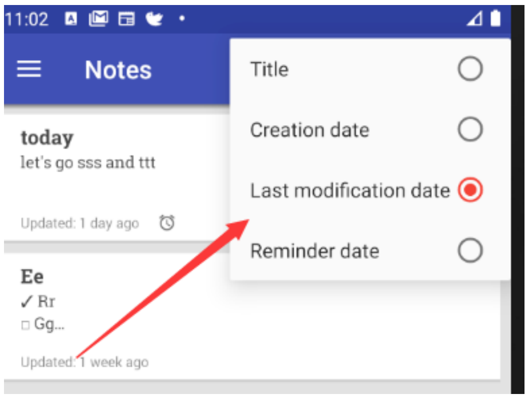
The Sorting function allows users to sort the notes in different ways. Users can sort the existing notes by title, creation date, last modification date and reminder date.
The reason why it's essential is that in order to provide a better user experience, the app must have different ways to help users find the notes they want as quick as possible, and Sorting is one of the most efficient ways to deal with it.

2.2 Behavioral Analysis

Basic modules of sorting are in the *ListFragment.java*. From Fig 2.1, we can see that the button of sorting submenu is in the Note-List interface. And the submenu is shown as in Fig 2.2.



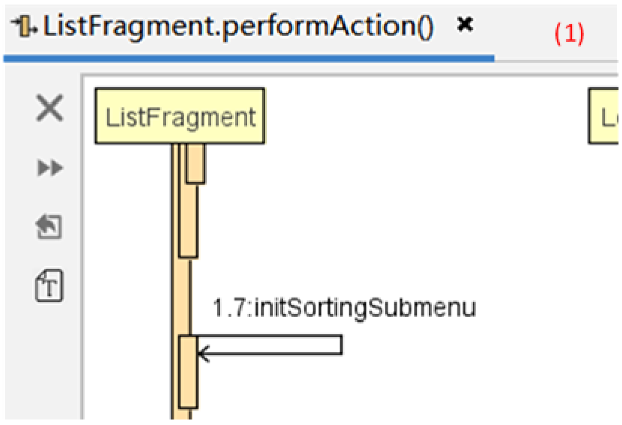
2-1.Locate the sorting submenu

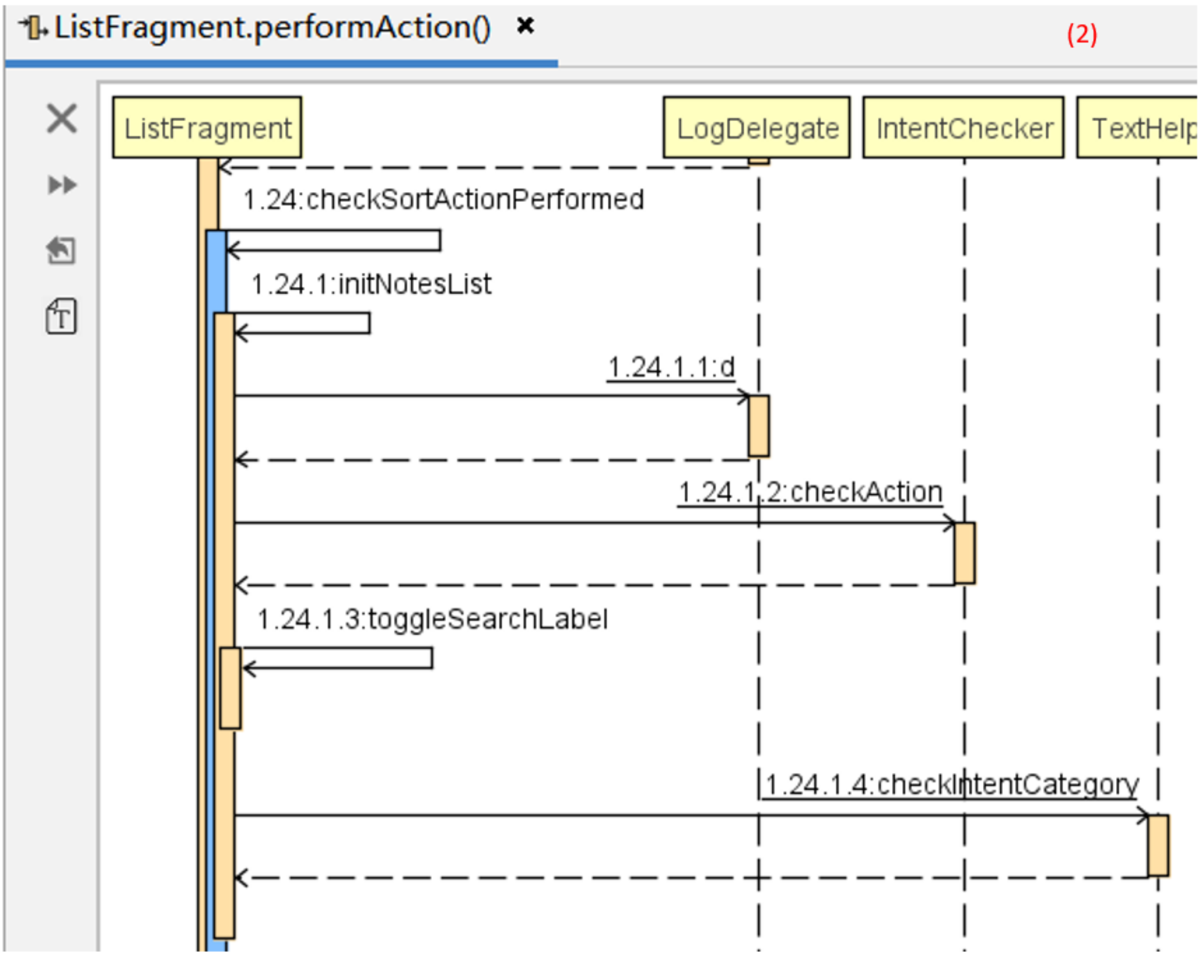


Title
Creation date
[✓ Last modification date]
Reminder date

2-2.Container of the sorting submenu

Because the ListFragment.java is used to handle all the actions in the Note List, and the main implementation of sorting in it.
So we searched through the *ListFragment.java* and we found **initSortingSubmenu ()**, which is used to initiate this submenu for different sorting methods. The sequence diagram based on **performAction()** shows that the **initSortingSubmenu ()** is called by **performAction()** (Fig 2-3-1). And in the sequence diagram, we found another method called **checkSortActionPerformed()**, the sorting operation is achieved in this method (Fig 2-3).

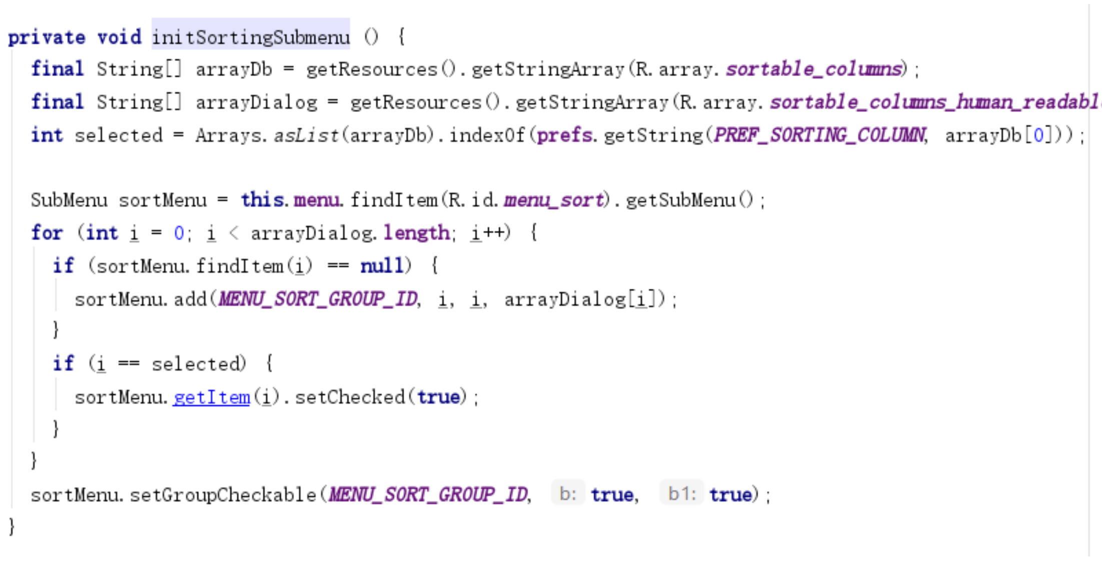



ListFragment.performAction()
(2)
ListFragment
LogDelegate
IntentChecker
TextHelp
1.24:checkSortActionPerformed
1.24.1:initNotesList
1.24.1.1:d
1.24.1.2:checkAction
1.24.1.3:toggleSearchLabel
1.24.1.4:checkIntentCategory

2-3.Sequence diagram starting from performAction method

2.3 Structural Analysis

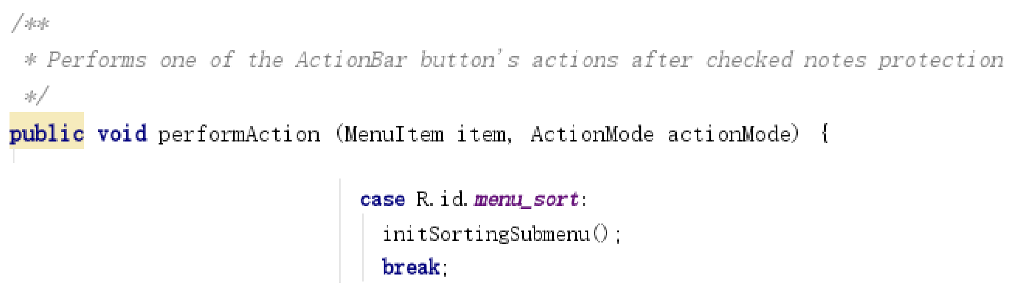
Before we sort the notes, we need to build the sorting submenu. So we need to analyze **initSortingSubmenu ()** first**.** In this module, it gives all items (sorting options) in this submenu a *group_id*: ***MENU_SORT_GROUP_ID,*** and sets the variable *selected* as the parameter ***PREF_SORTING_COLUMN***, which is saved in *sharedpreference*. If ***PREF_SORTING_COLUMN*** is null, *selected* will be set as the first item in the sorting submenu. The source code of **initSortingSubmenu ()** is shown in Fig.2-4.



```
private void initSortingSubmenu () {
  final String[] arrayDb = getResources().getStringArray(R.array.sortable_columns);
  final String[] arrayDialog = getResources().getStringArray(R.array.sortable_columns_human_readable);
  int selected = Arrays.asList(arrayDb).indexOf(prefs.getString(PREF_SORTING_COLUMN, arrayDb[0]));

  SubMenu sortMenu = this.menu.findItem(R.id.menu_sort).getSubMenu();
  for (int i = 0; i < arrayDialog.length; i++) {
    if (sortMenu.findItem(i) == null) {
      sortMenu.add(MENU_SORT_GROUP_ID, i, i, arrayDialog[i]);
    }
    if (i == selected) {
      sortMenu.getItem(i).setChecked(true);
    }
  }
  sortMenu.setGroupCheckable(MENU_SORT_GROUP_ID, b: true, b1: true);
}
```

2-4.Souce code of **initSortingSubmenu()**

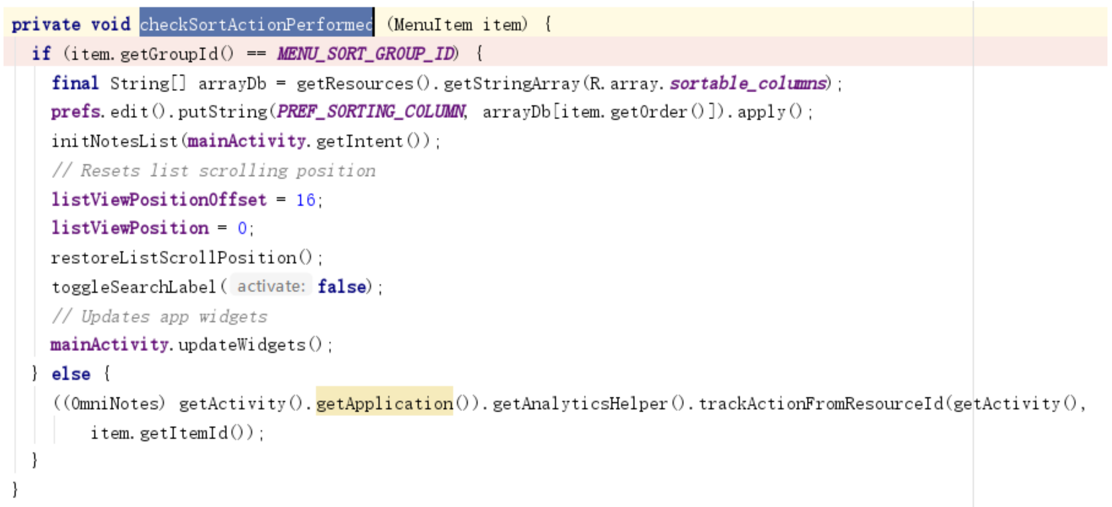
Then we use “find usage” to see where we call this method, and we found **performAction()** (Fig2-5), and according to the comment, this function is called when an ActionBar is pressed. And specifically, when the Sorting Menu ActionBar is pressed, it will switch to the corresponding case and invoke the **initSortingSubmenu()** method:



```
/**
 * Performs one of the ActionBar button's actions after checked notes protection
 */
public void performAction (MenuItem item, ActionMode actionMode) {

                        case R.id.menu_sort:
                          initSortingSubmenu();
                          break;
```

2-5.parts of **performAction()**

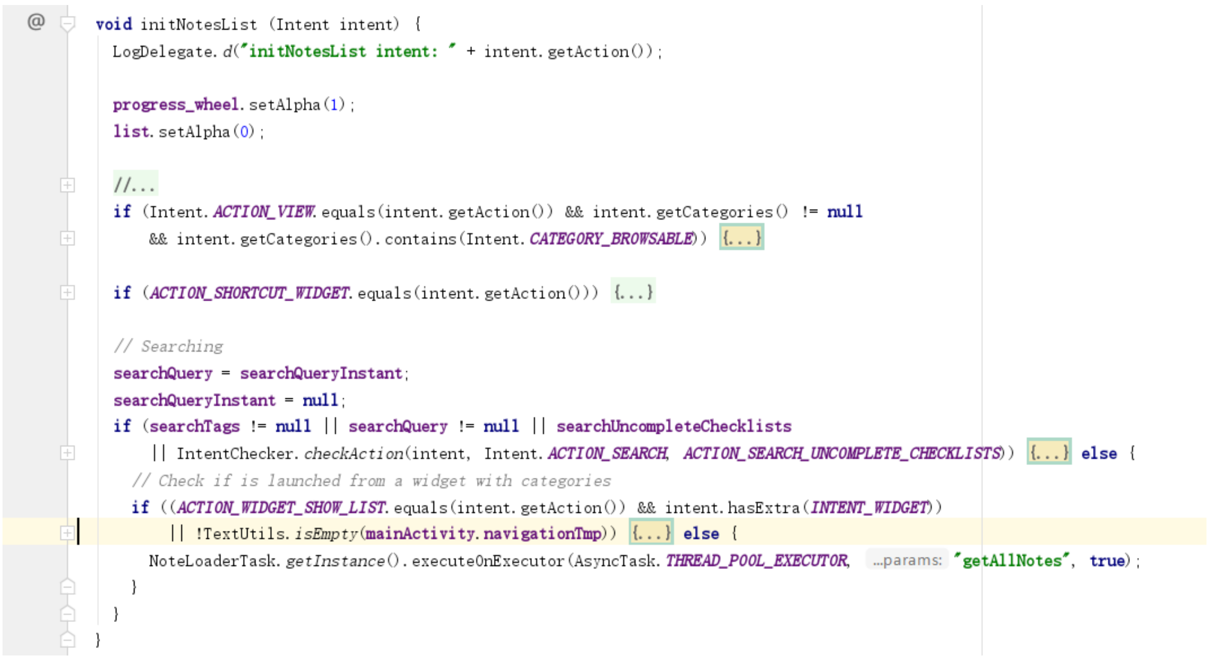
And according to the sequence diagram, we found another module named **checkSortActionPerformed()** (Fig 2-6)



```
private void checkSortActionPerformed (MenuItem item) {
  if (item.getGroupId() == MENU_SORT_GROUP_ID) {
    final String[] arrayDb = getResources().getStringArray(R.array.sortable_columns);
    prefs.edit().putString(PREF_SORTING_COLUMN, arrayDb[item.getOrder()]).apply();
    initNotesList(mainActivity.getIntent());
    // Resets list scrolling position
    listViewPositionOffset = 16;
    listViewPosition = 0;
    restoreListScrollPosition();
    toggleSearchLabel( activate: false);
    // Updates app widgets
    mainActivity.updateWidgets();
  } else {
    ((OmniNotes) getActivity().getApplication()).getAnalyticsHelper().trackActionFromResourceId(getActivity(),
        item.getItemId());
  }
}
```

2-6. Source code of **checkSortActionPerformed()**

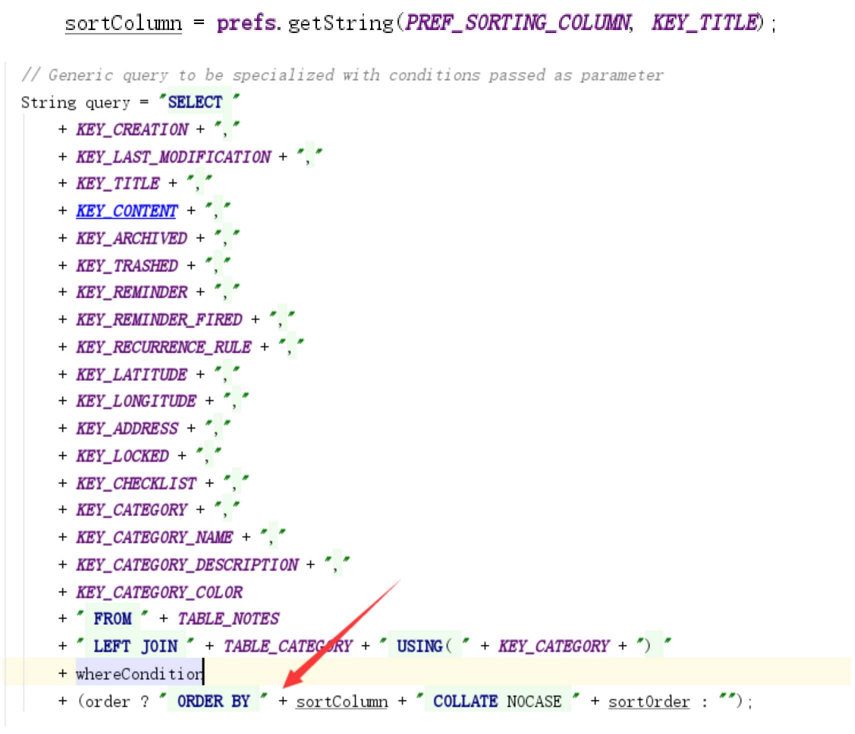
As the code shown in Fig 2-4, we gave the items in sorting submenu a group_id. Now we need that group_id to do further sorting operations in **checkSortActionPerformed()**.
In this method, ***PREF_SORTING_COLUMN*** is set as the name of the item we selected and then the whole note list is initiated. Because we use the items' names as the sorting order, we are going to see what happened in the **initNoteList()**.



```
@   void initNotesList (Intent intent) {
      LogDelegate.d("initNotesList intent: " + intent.getAction());

      progress_wheel.setAlpha(1);
      list.setAlpha(0);

      //...
      if (Intent.ACTION_VIEW.equals(intent.getAction()) && intent.getCategories() != null
          && intent.getCategories().contains(Intent.CATEGORY_BROWSABLE)) {...}

      if (ACTION_SHORTCUT_WIDGET.equals(intent.getAction())) {...}

      // Searching
      searchQuery = searchQueryInstant;
      searchQueryInstant = null;
      if (searchTags != null || searchQuery != null || searchUncompleteChecklists
          || IntentChecker.checkAction(intent, Intent.ACTION_SEARCH, ACTION_SEARCH_UNCOMPLETE_CHECKLISTS)) {...} else {
        // Check if is launched from a widget with categories
        if ((ACTION_WIDGET_SHOW_LIST.equals(intent.getAction()) && intent.hasExtra(INTENT_WIDGET))
            || !TextUtils.isEmpty(mainActivity.navigationTmp)) {...} else {
          NoteLoaderTask.getInstance().executeOnExecutor(AsyncTask.THREAD_POOL_EXECUTOR, ...params: "getAllNotes", true);
        }
      }
    }
```

2-7.Source code of **initNoteList()**

initNoteList() is a big method and most of the contents are about the searching function. At the end of the module, the **NoteLoaderTask** executed a function called **getAllNotes** (NoteLoaderTask will be explained in section 2.4.2).
The getAllNotes() is implemented in DBHelper class and the final step of reference is the getNotes(). In this method, the **PREF_SORTING_COLUMN** saved in *sharedpreference* is used as the order. So with the query built, we can get a result set with the order we selected in sorting submenu, rebuild the whole list, and replace the old list with the new one. That's how the sorting function works.



```
sortColumn = prefs.getString(PREF_SORTING_COLUMN, KEY_TITLE);

// Generic query to be specialized with conditions passed as parameter
String query = "SELECT "
    + KEY_CREATION + ","
    + KEY_LAST_MODIFICATION + ","
    + KEY_TITLE + ","
    + KEY_CONTENT + ","
    + KEY_ARCHIVED + ","
    + KEY_TRASHED + ","
    + KEY_REMINDER + ","
    + KEY_REMINDER_FIRED + ","
    + KEY_RECURRENCE_RULE + ","
    + KEY_LATITUDE + ","
    + KEY_LONGITUDE + ","
    + KEY_ADDRESS + ","
    + KEY_LOCKED + ","
    + KEY_CHECKLIST + ","
    + KEY_CATEGORY + ","
    + KEY_CATEGORY_NAME + ","
    + KEY_CATEGORY_DESCRIPTION + ","
    + KEY_CATEGORY_COLOR
    + " FROM " + TABLE_NOTES
    + " LEFT JOIN " + TABLE_CATEGORY + " USING( " + KEY_CATEGORY + ") "
    + whereCondition
    + (order ? " ORDER BY " + sortColumn + " COLLATE NOCASE " + sortOrder : "");
```

2-8.Key source code of **getNotes()**

2.4 Related classes

2.4.1 DBHelper.java

The DBHelper class is used to conduct database related operations. In fact, note sorting itself is to get all the note information from the database and return a note list in an order we want. So the DBHelper class is necessary through note sorting. It can be used to get all notes, uncategorized notes and untrashed notes, etc.

2.4.2 NoteLoaderTask.java

The NoteLoaderTask class is responsible for loading the notes from the database. It extends AsyncTask class. We use a method's name(e.g. getAllNotes) in DBHelper as a parameter and the executor will call the corresponding method in DBHelper and run the database-operation method asynchronously.