Turbo Editor

Jean Amirian 6903460,

Abdenour Djafri 5653223,

Ayman Alshehri 4796810,

Julian Wolfe 6625177,

Alain Trinh 6638260

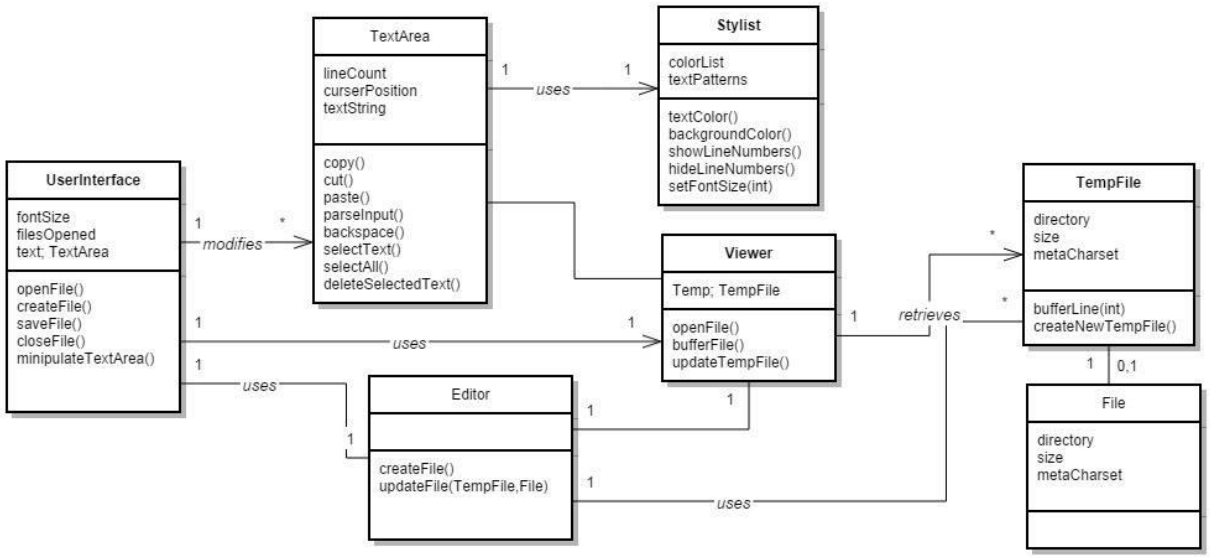
We have compiled and assessed this project a believe it to be of a reasonable size for a term project.

Summary of Project

Turbo Editor is a simple, yet powerful, text editor on the Android platform which enables users to directly focus on the job at hand. The editor is capable of text editing as well as code editing and should surely come in handy for individuals whose lives are on the go and work doesn't stop following. Considering this application will likely serve a purpose to the members of this team, offer an experience developing on the android platform and is written in Java, a language all members of the team are relatively fluent in, this project serves as a fulfilling contribution to the open source community.

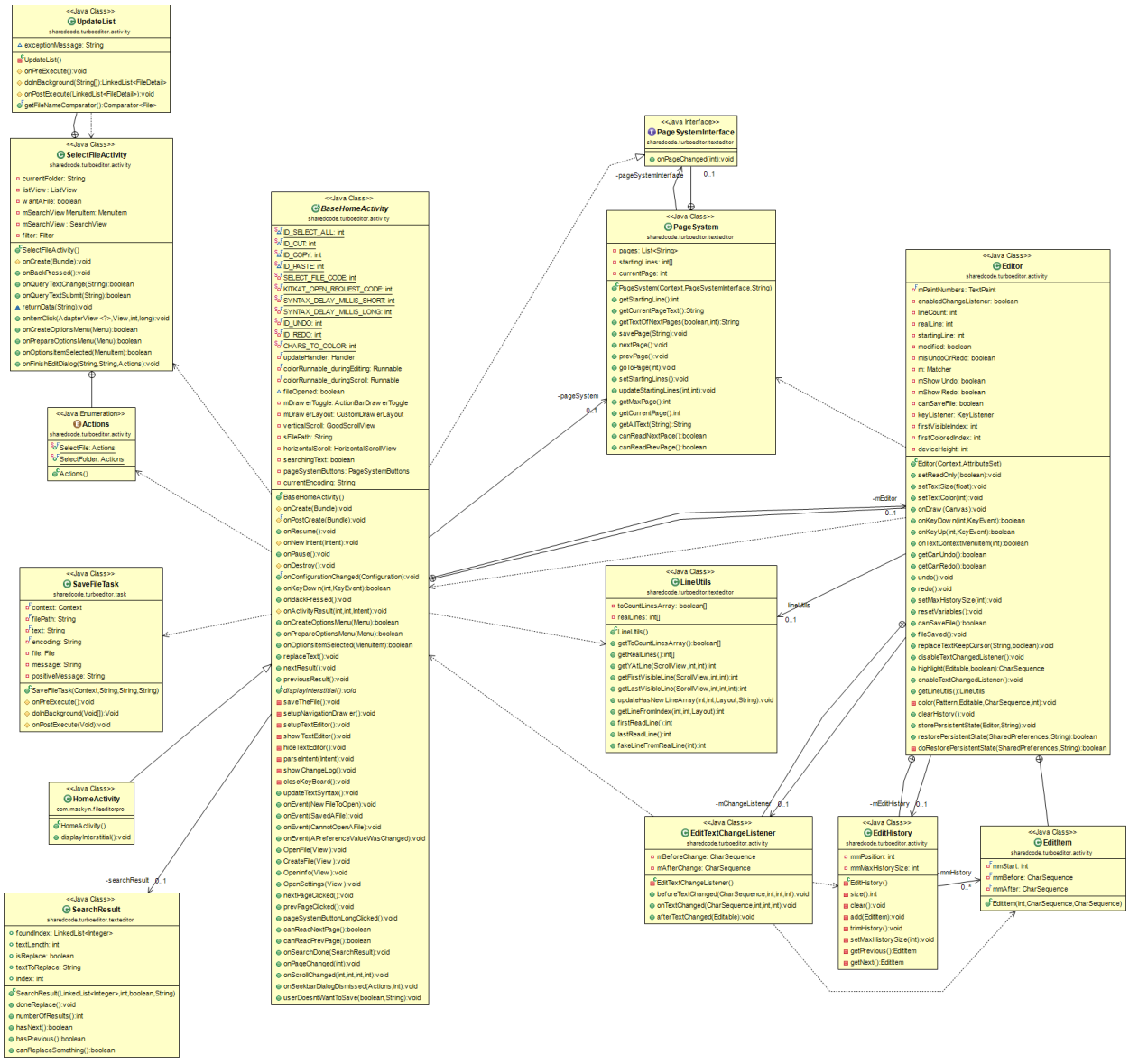
Class Diagram of Actual System

Conceptual Diagram



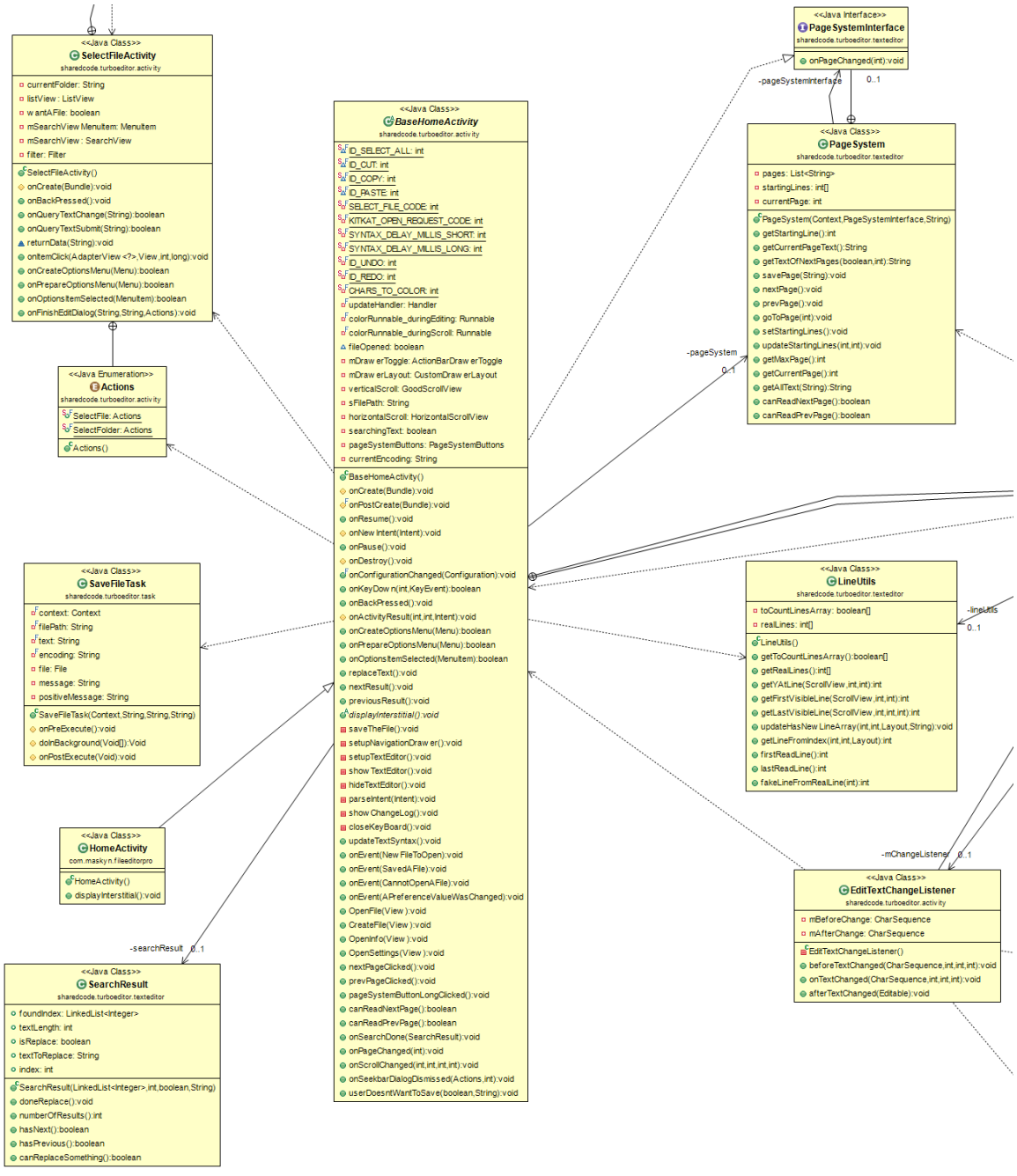
Actual System

The following diagram shows an overall view of the part of the system we will be working with. The overview shows the interactions between *BaseHomeActivity*, *Editor* and their various dependencies.



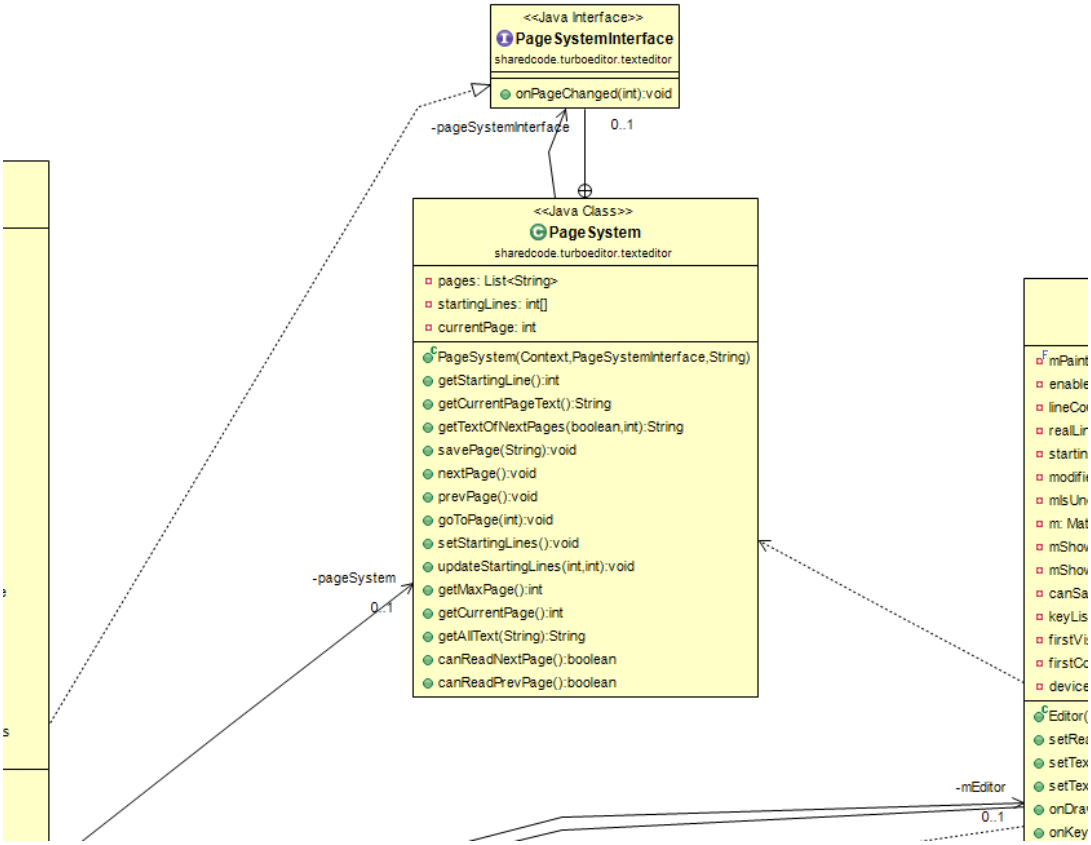
Here we zoom in on the *BaseHomeActivity* class. This class currently:

- Manages the *Editor*, which displays the text
- Launches the *SelectFileActivity* to select a file to edit
- Saves files using the *SaveFileTask*
- Handles in-document searching with the *SearchResult* class
- Manages document pages using the *PageSystem* by implementing the *PageSystemInterface*
- Depends on *LineUtils* to manage document line numbers
- Is depended on by the *EditTextChangeListener* which listens for changes in the document



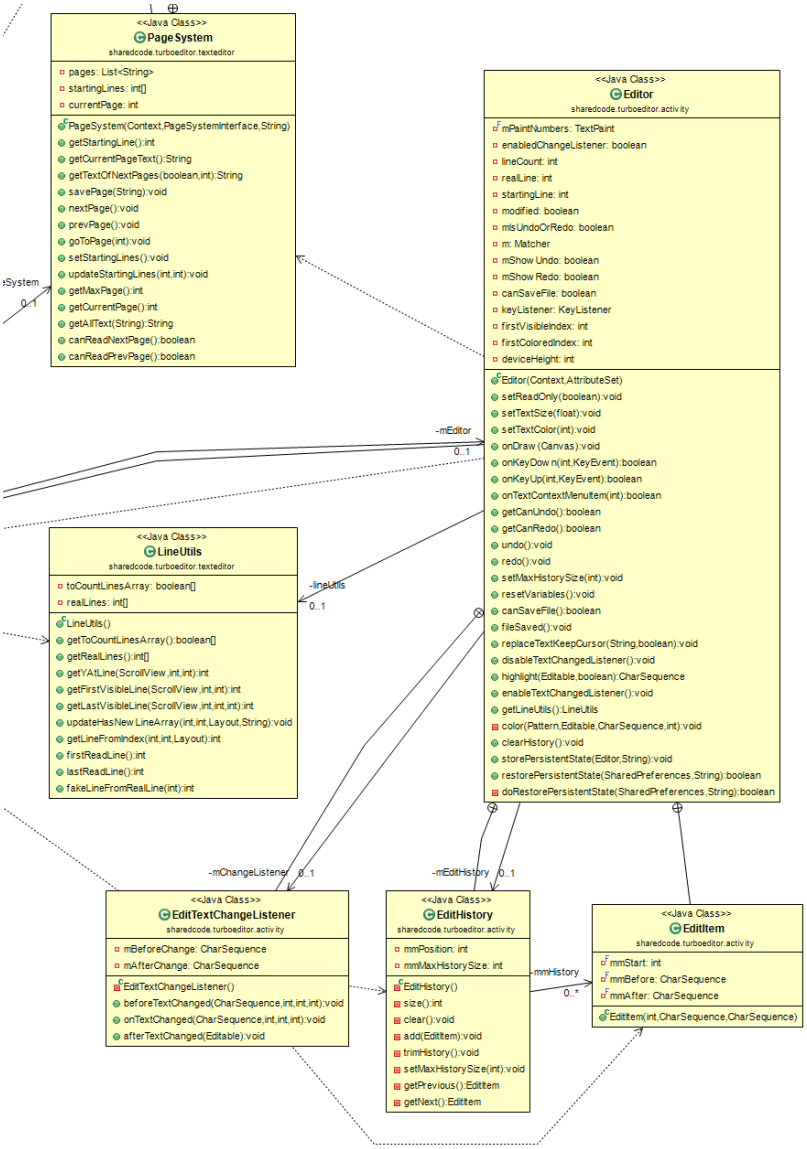
Here we zoom into the *PageSystem* class. The *PageSystem* currently:

- Stores document pages as a *List* of strings
- Is managed by the *BaseHomeActivity*
- Defines a *PageSystemInterface* that the *BaseHomeActivity* implements, to listen for *onPageChanged* events



Here we zoom into the *Editor* class and the classes it interacts with. The *Editor* currently:

- Extends *EditText* from the Android SDK, which provides a UI element in which to edit text
- Depends on the *PageSystem* to obtain metrics about text lines
- Defines an *EditTextChangeListener* to listen for changes in the document
- Defines an *EditHistory* class which keeps a history of changes for undo and redo operations
- Defines an *EditItem* class, used by *EditHistory*, which encapsulates a particular edit item



The actual system differs greatly from the conceptual system as the author has a significantly different view on how the system should be implemented. A point to consider is the fact that the author has done some major refactorings himself in the past weeks since we chose the project. While the quality of code has improved from the previous revision, it remains obvious that the architecture is not final. Also, from the size of several classes and the tight coupling between them, we can presume that the original author is not very experienced in object-oriented design.

In order to analyze the codebase and to generate our diagrams, we used ObjectAid on a stub project into which the source files were copied. We could not have used the original project tree as the project is developed using Android Studio, and ObjectAid is an Eclipse extension. It was sufficient to analyze a stub version of the project as we only needed to see the relationship between classes.

The following code shows the methods and attribute declarations we will work on, along with a brief explanation of how the methods work and interact with other classes.

```
1.  public abstract class BaseHomeActivity extends Activity implements FindTextDialog
2.          .SearchDialogInterface, GoodScrollView.ScrollInterface, PageSystem.PageSystemInterface,
3.          PageSystemButtons.PageButtonsInterface, SeekbarDialog.ISeekbarDialog, SaveFileDialog.ISaveDialog {
4.
5.      // other members...
6.      private Editor mEditor;
7.
8.      /**
9.       * Save the file before pausing the Activity. Checks the state of Editor mEditor.
10.      */
11.     @Override
12.     public void onPause() { /* ... */ }
13.
14.     /**
15.      * Interprets a key code and sends it to the Editor mEditor if necessary.
16.      */
17.     @Override
18.     public boolean onKeyDown(int keyCode, KeyEvent event) { /* ... */ }
19.
20.     /**
21.      * Takes the appropriate actions when the back button is pressed. If the app is about to exit,
22.      * prompt the user to save the file before exiting.
23.      */
24.     @Override
25.     public void onBackPressed() { /* ... */ }
26.
27.     /**
28.      * Sets menu options from state, including Editor mEditor's state.
29.      */
30.     @Override
31.     public boolean onPrepareOptionsMenu(Menu menu) { /* ... */ }
32.
33.     /**
34.      * Performs actions requested by selected menu item. Some of these interact with Editor mEditor.
35.      */
36.     @Override
37.     public boolean onOptionsItemSelected(MenuItem item) { /* ... */ }
38.
39.     //endregion
40.
41.     // region OTHER THINGS
42.
43.     /**
44.      * Replaces the found instances of the search with the specified replacement, in Editor mEditor.
45.      */
46.     public void replaceText() { /* ... */ }
47.
48.     /**
49.      * Sets the text selection in Editor mEditor to the next found result.
50.      */
51.     public void nextResult() { /* ... */ }
```

```
52.
53.     /**
54.      * Sets the text selection in Editor mEditor to the previous found result.
55.      */
56.     public void previousResult() { /* ... */ }
57.
58.     /**
59.      * Saves the file using the text retrieved from Editor mEditor.
60.      */
61.     private void saveTheFile() { /* ... */ }
62.
63.     /**
64.      * Acquires handles to the views, and decorates Editor mEditor with appropriate scroll views.
65.      */
66.     private void setupTextEditor() { /* ... */ }
67.
68.     /**
69.      * Shows the Editor mEditor and sets appropriate state.
70.      */
71.     private void showTextEditor() { /* ... */ }
72.
73.     /**
74.      * Hides the Editor mEditor and sets appropriate state.
75.      */
76.     private void hideTextEditor() { /* ... */ }
77.     // closes the soft keyboard
78.     private void closeKeyBoard() throws NullPointerException {}
79.
80.     /**
81.      * Uses Editor mEditor to check if there is an active text selection. If so, do not update
82.      * the syntax highlighting.
83.      */
84.     public void updateTextSyntax() {
85.         if (!PreferenceHelper.getSyntaxHiglight(getBaseContext()) || mEditor.hasSelection() ||
86.                 updateHandler == null || colorRunnable_duringEditing == null)
87.             return;
88.         // ...
89.     }
90.
91.     /**
92.      * Uses Editor mEditor to check if the file can be saved, and if so, retrieves the text from
93.      * mEditor.
94.      */
95.     public void onEvent(final EventBusEvents.NewFileToOpen event) {
96.         if (fileOpened && mEditor.canSaveFile()) {
97.             SaveFileDialog.newInstance(sFilePath, pageSystem.getAllText(mEditor
98.                     .getText().toString()), currentEncoding, true,
    event.getFile().getAbsolutePath()).show(getFragmentManager(),
99.                     "dialog");
100.            return;
101.        }
102.        // ...
103.    }
104.
105.    /**
106.     * Clears history and marks file as saved on Editor mEditor.
107.     */
108.    public void onEvent(EventBusEvents.SavedAFile event) {
109.        mEditor.clearHistory();
110.        mEditor.fileSaved();
111.        // ...
112.    }
113.
114.    public void onEvent(EventBusEvents.CannotOpenAFile event) {}
115.
116.    /**
117.     * Reconfigures internal state by either reassigning Editor mEditor to different views, or by
118.     * calling mutators on mEditor to change its internal state. Should have each branch extracted
119.     * into separate methods for clarity.
120.     */
121.    public void onEvent(EventBusEvents.APreferenceValueWasChanged event) {
122.        // giant block of if-else statements to handle each
123.    }
124.
125.    //region Ovverideses
126.
```

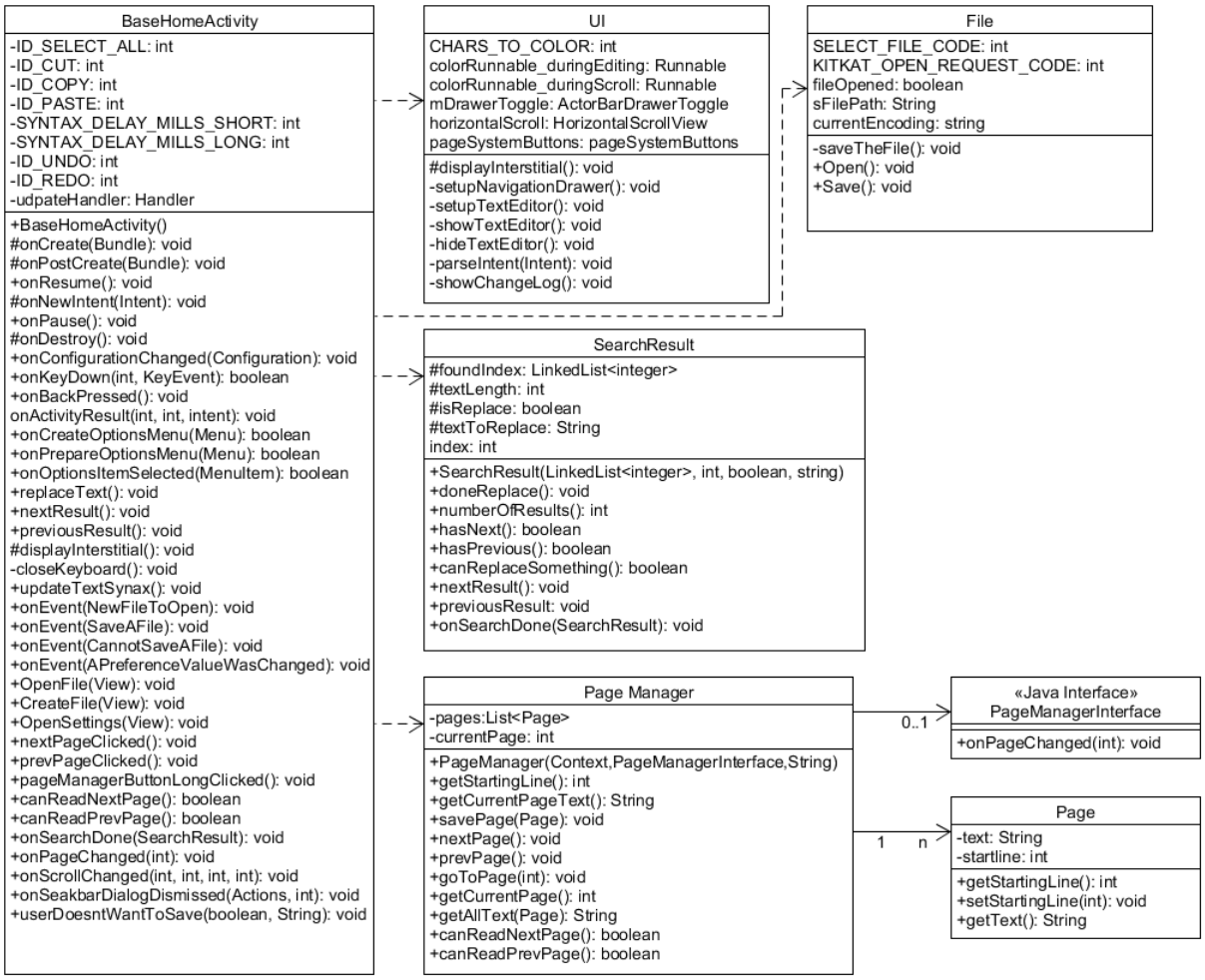
```
127.     /**
128.      * Retrieves text from Editor mEditor to save the page, and then switches to the next page.
129.      */
130.     @Override
131.     public void nextPageClicked() {
132.         pageSystem.savePage(mEditor.getText().toString());
133.         // ...
134.     }
135. 
136.     /**
137.      * Retrieves text from Editor mEditor to save the page, and then switches to the previous page.
138.      */
139.     @Override
140.     public void prevPageClicked() {
141.         pageSystem.savePage(mEditor.getText().toString());
142.         // ...
143.     }
144. 
145.     /**
146.      * Sets the selection to the first search result and scrolls there by calling the appropriate
147.      * methods on Editor mEditor.
148.      */
149.     @Override
150.     public void onSearchDone(SearchResult searchResult) { /* ... */ }
151. 
152.     /**
153.      * Cleans up state, including clearing history on the Editor mEditor.
154.      */
155.     @Override
156.     public void onPageChanged(int page) { /* ... */ }
157. 
158.     /**
159.      * Sets correct syntax highlighting handler during scroll. Checks if Editor mEditor has
160.      * a selection.
161.      */
162.     @Override
163.     public void onScrollChanged(int l, int t, int oldl, int oldt) { /* ... */ }
164. 
165. 
166.     /**
167.      * Depending on the action taken, either change page in the Editor mEditor or scroll to the
168.      * specified line.
169.      */
170.     @Override
171.     public void onSeekbarDialogDismissed(SeekbarDialog.Actions action, int value) { /* ... */ }
172. 
173.     /**
174.      * Unsets canSaveFile on Editor mEditor if user doesn't want to save.
175.      */
176.     @Override
177.     public void userDoesntWantToSave(boolean openNewFile, String pathOfNewFile) { /* ... */ }
178. 
179. 
180.     public static class Editor extends EditText {
181.         /**
182.          * Constructor.
183.          */
184.         public Editor(final Context context, AttributeSet attrs) { /* ... */ }
185. 
186.         /**
187.          * Sets the size of the text as well as the line numbers, which are smaller.
188.          */
189.         @Override
190.         public void setTextSize(float size) { /* ... */ }
191. 
192.         /**
193.          * Sets the color of the text, as well as the line numbers.
194.          */
195.         @Override
196.         public void setTextColor(int color) { /* ... */ }
197. 
198.         /**
199.          * Draws the Editor.
200.          */
201.         @Override
202.         public void onDraw(final Canvas canvas) { /* ... */ }
```

```
203.
204.        //region Other
205.
206.        /**
207.         * Handles the given key code. If ctrl is pressed, perform appropriate shortcut actions.
208.         * If tab is pressed, insert an indent. Otherwise, just pass the event to superclass
209.         * method.
210.         */
211.        @Override
212.        public boolean onKeyDown(int keyCode, KeyEvent event) { /* ... */ }
213.
214.        /**
215.         * Handles key up events.
216.         */
217.        @Override
218.        public boolean onKeyUp(int keyCode, KeyEvent event) { /* ... */ }
219.
220.        /**
221.         * Handles context menu events.
222.         */
223.        @Override
224.        public boolean onTextContextMenuItem(final int id) { /* ... */ }
225.
226.        /**
227.         * Returns whether undo can be performed.
228.         */
229.        public boolean getCanUndo() { /* ... */ }
230.
231.        /**
232.         * Returns whether redo can be performed.
233.         */
234.        public boolean getCanRedo() { /* ... */ }
235.
236.        /**
237.         * Performs undo.
238.         */
239.        public void undo() { /* ... */ }
240.
241.        /**
242.         * Performs redo.
243.         */
244.        public void redo() { /* ... */ }
245.
246.        /**
247.         * Set the maximum history size.
248.         */
249.        public void setMaxHistorySize(int maxHistorySize) { /* ... */ }
250.
251.        /**
252.         * Resets internal state.
253.         */
254.        public void resetVariables() { /* ... */ }
255.
256.        /**
257.         * Returns whether the file can be saved. Should not be in Editor.
258.         */
259.        public boolean canSaveFile() { /* ... */ }
260.
261.        /**
262.         * Returns whether the file has been saved. Should not be in Editor.
263.         */
264.        public void fileSaved() { /* ... */ }
265.
266.        /**
267.         * Replaces text in the Editor with given text. If highlighting is enabled,
268.         * perform the highlighting at the same time.
269.         *
270.         * This method is responsible for way too much.
271.         */
272.        public void replaceTextKeepCursor(String textToUpdate, boolean mantainCursorPos) {
273.            // ...
274.        }
275.
276.        /**
277.         * Disables the text change listener.
278.         */
```

```
        public void disableTextChangedListener() { /* ... */ }

        /**
         * Returns a highlighted version of the given text. If newText is set, highlight the
         * given text, otherwise highlight the visible portion of the Editor.
         *
         * This method actually checks the type of the file by getting its extension.
         * Let's fix that.
         */
        public CharSequence highlight(Editable editable, boolean newText) { /* ... */ }

        /**
         * Enables the text change listener.
         */
        public void enableTextChangedListener() { /* ... */ }

        /**
         * Returns the LineUtils object for this Editor.
         */
        public LineUtils getLineUtils() { /* ... */ }

        /**
         * Applies color to the given text.
         *
         * This method checks the pattern type and applies the appropriate color. There is probably
         * a better way to do this.
         */
        private void color(Pattern pattern,
                           Editable allText,
                           CharSequence textToHighlight,
                           int start) { /* ... */ }

        /**
         * Clears history.
         */
        public void clearHistory() { /* ... */ }

        /**
         * Stores preferences.
         */
        public void storePersistentState(
                SharedPreferences.Editor editor,
                String prefix) { /* ... */ }

        /**
         * Restores preferences.
         */
        public boolean restorePersistentState(
                SharedPreferences sp, String prefix)
                throws IllegalStateException { /* ... */ }

        /**
         * Actually restores preferences. Previous method wraps this one.
         */
        private boolean doRestorePersistentState(
                SharedPreferences sp, String prefix) { /* ... */ }
    }
}
```

Code Smells and System Level Refactorings

The *BaseHomeActivity* class will be refactored to remove large class code smell and separate responsibilities to increase cohesion. The *BaseHomeActivity* should only focus on the lifecycle of the app and not the underlying functionality that corresponds to it. This will create 2 new classes and alter an existing class. The new *File* class will acquire the attributes and methods in the *BaseHomeActivity* class that deal with file manipulation and interactions with the existing *SaveFileTask* and *SelectFileActivity* (not included in the refactored UML diagram below). The new *UI* class will acquire all attributes and functions relating to the views that are returned on screen for the user, along with any functions that handle the presentation of dialogs the app uses to gauge user input. Search-related functions will be placed in the existing *SearchResult* class, which handles searches inside the app. To prevent breaking the use of the *BaseHomeActivity* interface, any public and protected functions will simply map to the new functions created in the new specialized classes. All private functions will simply be moved to the new classes and removed from the central class. The following UML diagrams illustrate the structural changes. In an ideal refactoring, many of the duplicated public and protected functions in *BaseHomeActivity* would become exclusive to the new classes, but in reality the *BaseHomeActivity* will become a wrapper to the new classes.



Inside the *BaseHomeActivity* class, the duplicated code within conditional statements in the onEvent(EventBusEvents.APreferenceValueWasChanged event) method will be extracted to new methods to clarify functions, remove comments and remove duplicated code.

Feature envy will be reduced in the *Editor* class, where its methods currently handle responsibilities that should be handled by other classes. The *Editor* class should be responsible for modifying text when needed and includes methods such as setTextSize, setTextColor to do so. However, it also has methods like onKeyUp and onKeyDown, which handle events when it shouldn't. It also unnecessarily handles file operations and should not contain an attribute for determining whether a file can save.

The onKeyUp function inside the *Editor* class uses 7 different switch statements that return the same value. The switch statements can be replaced by a single if statement, checking all seven keyEvents. The code inside the method is shorter and optimized, hence, using less resources.

Remove an instance of Primitive Obsession in the *PageSystem* class by creating a *Page* class that deals with its text, highlighting and startLine attributes. The *Page* will replace the need for a List of Strings in the *PageSystem*.

Rename *PageSystem* to *PageManager* to accurately reflect the specialization of the class and remove unnecessary comment code smell.

Specific Refactorings that you will implement in Milestone 4

For milestone 4, we intend to complete:

- The reduction of feature envy in the *Editor* class
- The replacement of switch statements with an equivalent if statement in the onKeyUp method
- The extraction of a *Page* class from *PageSystem*, to separate responsibilities
- The renaming of *PageSystem* to *PageManager* to more accurately reflect the purpose of the class
- The removal of code duplication within the *BaseHomeActivity* class by extracting a method from the duplicated lines of code, and replacing all instances of the duplication by a call to the new method

One of the specific refactorings that will be implemented will tackle duplication of code within a specific method in the *BaseHomeActivity* class.

To increase self-documentation, the body of the if statements within the onEvent(EventBusEvents.APreferenceValueWasChanged event) method inside the *BaseHomeActivity* class will be extracted into new methods with self-explanatory names. This will also reduce code duplication found frequently within the method.

```
1.  if (PreferenceHelper.getWrapContent(getBaseContext())) {
2.      horizontalScroll.removeView(mEditor);
3.      verticalScroll.removeView(horizontalScroll);
4.      verticalScroll.addView(mEditor);
5.  } else {
6.      verticalScroll.removeView(mEditor);
7.      verticalScroll.addView(horizontalScroll);
8.      horizontalScroll.addView(mEditor);
9.  }
```

The code above sets the proper decorators on *mEditor* depending on whether text wrapping is enabled or not. If wrapping is enabled, there is no possibility for horizontal scrolling, so the *horizontalScroll* decorator is removed. Otherwise, the *horizontalScroll* decorator is added.

This snipped was found multiple times in the *BaseHomeActivity* class and would be best implemented as a private method considering how often it is used.

To fix the code duplication, we will move the above snippet into its own method, called setScrollDecorators(). Next, we will replace all instances of this snippet with a call to the new method. Finally, we will do a functional test to ensure that the refactoring has not broken any functionality. Unfortunately, there no zero unit test coverage in this project, so we cannot automate testing.