For the Fun of It: Writing Your Own Text Editor, Part 2

Using Swing to write a GUI for the text editor

by Ian Darwin

November 21, 2019

In the first part of this series, I covered designing and building a simple line-mode text editor, edj. In this article, I transform that editor into a graphical screen editor called vedj (visual edj) while keeping the existing command-line support. Why? First, Java on the desktop has more life in it than you probably expect. Plus, getting out of your “comfort zone” can move your focus towards interesting design and implementation issues. (The code for both editors is available for download on GitHub.)

The project is destined to remain a text editor, that is, a program for changing a plain-text file. Plain text—text without font or color changes—still makes up a vast portion of the world’s computer-facing information: batch/script files, program source files, configuration files, log files, and much more. There’s a big jump in complexity from the plain-text editor presented here to a full word processor that can select typefaces, type size, and color; embed images and spreadsheets; align text left, right, or center; and so much more. If you find you need to build that kind of tool using Swing, I recommend the O’Reilly book Java Swing by Eckstein, Loy, and Wood. (This is an older book. I still refer to it because Swing has been spared the rapid pace of change of some other parts of Java. Also, I prefer writing GUIs in Swing. Similar books exist for those who prefer JavaFX.) But be warned: People with more money and resources than you have tried building a Microsoft Office–like suite in Java before and never made it out of beta.

In lieu of a formal design document, here’s a very quick summary of the desired feature set:

<table>
<thead>
<tr>
<th>Feature &quot;Screen Editor&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>As a user of the screen mode, I want to be able to:</td>
</tr>
<tr>
<td>- click anywhere and start typing</td>
</tr>
<tr>
<td>- select a range of text via the mouse, and:</td>
</tr>
<tr>
<td>type over it;</td>
</tr>
<tr>
<td>copy it;</td>
</tr>
<tr>
<td>cut it;</td>
</tr>
<tr>
<td>delete it.</td>
</tr>
<tr>
<td>- Select an insertion point or a range of text, and paste cut/copied text in that location or in place of that text</td>
</tr>
<tr>
<td>As a user of the line mode, I want to be able to:</td>
</tr>
<tr>
<td>- do anything I could do in edj.</td>
</tr>
</tbody>
</table>

The buffer primitives and command handling were the focus of the previous article. In this article, I focus on the GUI. The two main Java GUI packages these days are Swing and JavaFX. I chose Swing because it’s more familiar to most readers; I suspect the logic within the code would not be much different for JavaFX.
Swing and the GUI Bits

For those who don’t know Swing, here’s a brief overview. The main parts of the GUI code are in the javax.swing package, and many are subclassed from javax.swing.JComponent. You use a javax.swing.JFrame for an application’s main window; javax.swing.JPanel and other composite components for layouts; and javax.swing.JButton, javax.swing.JLabel, javax.swing.JList, javax.swing.JTextArea, and so on for individual components.

For most of these items, there is not much to think about. There’s a menu bar at the top with the familiar File/Edit/View menus, a large panel in the middle to display the current in-memory file, and, in this case, a place to type commands from the command-line editor. This facility should include a recall mechanism so you don’t have to retype a command if you make an error or if you simply want to rerun something.

Finally, you should also be able to make changes to the text by selecting a range of text. This range may be anything from one character, to a word, to multiple lines, including parts of lines at the beginning and end, up to the whole file. Once text is selected, you should be able to cut or copy it for later pasting, type new text to replace it, or click Delete to discard it. This is the area that causes the most work when this type of GUI is layered on top of a line-oriented editor.

The main panel could be implemented in several ways in Swing. You could use a javax.swing.JList or a javax.swing.JTextArea to get some functionality. Or you could use a blank canvas and do all the work yourself. Each approach has some benefits. The javax.swing.JList approach allows each line to be a separate object, which would map well to my previous BufferPrims implementation (which used a java.util.List<String> to control the lines). However, it does not allow the selection of partial lines without some complicated mouse coordinate-to-character mapping. And selecting partial lines is a fairly basic function of a screen-based text editor. Writing that capability would significantly increase the size of the code, be complex, and be error-prone.

The javax.swing.JTextArea approach has more flexibility, but it requires care in handling the interaction between the logic in javax.swing.JTextArea and the logic in this application. The javax.swing.JCanvas class doesn’t actually exist, so the choice would probably be a javax.swing.JPanel or a javax.swing.JComponent. Here, I would draw the lines of text directly from the buffer in a paint-type method and handle the complexity of text selections and keyboard events entirely in the editor.

I started implementing several of these approaches. In the end, I went with a javax.swing.JTextArea and a custom implementation of BufferPrims called BufferPrimsJText to handle the mapping between the two views of the text buffer. javax.swing.JTextArea basically holds the text as a long String, but a helper class, Swing’s java.util.Document class, knows where the line boundaries are. The code in this editor doesn’t use the java.util.Document interface directly, but a full word processor would.

The BufferPrimsJText code moves text between the line-oriented buffer and the javax.swing.JTextArea. I rely on the javax.swing.JTextArea to maintain the text buffer contents. Almost all state information (even getting the line count) is obtained dynamically by querying the javax.swing.JTextArea.

To illustrate the general working of this interface, here is the replace() method from BufferPrimsJText. This method would be invoked when the user types a substitute command such as /Hello/Goodbye/g in the command window. This example says to replace all (g at the end stands for global) Hello strings with Goodbye in the current line. Without the g the method would replace only the first occurrence in each line.

```java
class BufferPrimsJText {  
    public void replace(String oldRE,  
                        String replacement, boolean all  
                        ) {  
        try {  
            final int adjustedLineNum =  
                lineNumToIndex(getCurrentLineNumber());  
            int startRange =  
```
This code starts by calling my `lineNumToIndex()` method to map the line number from the one-based lines in the editor's view of the text buffer (as discussed in the previous article) to the zero-based lines of the `JTextArea` Document. Then the `JTextArea` methods are used to compute the character index of the text in the current line (subtracting one so I don't get the \n character at the line end). This information is used to grab the actual string representing the current line. Then one of the standard `String` methods, `replaceAll()` or `replaceFirst()`, is chosen, based on the `all` parameter (which in turn comes from the presence or absence of the `g` at the end of the command). Finally, the modified string is pushed back into the `JTextArea` by the latter's `replaceRange()` method.

The other `BufferPrimsJText` methods work similarly: Compute a range, and then tell the `JTextArea` to perform the required operation on it.

**What About Undo?**

At first, I thought I would need to do some complicated mapping involving selection events to handle events such as the user selecting part of a line and then deleting it. However, it turned out to be easier: My `BufferPrimsJText` simply calls code in the `JTextArea` to do the work. And if my code updates the buffer (as my `replace()` method does), Swing still considers that an undoable event, and it calls my `UndoableEventListener`. I merely pass that along to an instance of Swing's own `UndoManager`, which provides all the `undo/redo` capability.

I had to provide only a pair of Swing `Action` classes to interface with my code and keep the `Undo/Redo` menu items enabled at the appropriate time.

One minor glitch is that a `JTextArea.replaceRange()` operation is recorded as two actions: a deletion followed by an insertion. As a result, the user would have to select `Undo` twice to undo what seems like a single action. Otherwise, it just works.

**Test Early and Often**

Every treatise on testing GUI code basically says not to test, meaning don't test the GUI code but instead, keep it so simple that it doesn't need testing. That is, of course, the purpose of my `BufferPrims` interface and the `Commands` class, which keep line-manipulation logic separate from the user interface code. There are, as you now know from the section "Swing and the GUI Bits" above, several different `BufferPrims` implementations. When you have multiple implementations of an interface, it's usually best to test all of them against the same set of tests. But please don't do that by copying and pasting tests from one test class into another! This is a perfect use case for automation.

For this project, I used JUnit 4's `Parameterized` test runner. Test classes so annotated provide a `@Parameters-annotated` method that returns a list of arrays of `Object` and a constructor method that accepts the contents of one of those arrays. When there is only one parameter to the constructor, as here, the annotated method just returns an array of the parameter values. JUnit will call the constructor and then run all the
The BufferPrimsTest class uses reflection (that is, 
Class.getConstructor().newInstance()) to create the target 
BufferPrims implementation being tested. However, the JTextArea 
version of BufferPrims has a different constructor argument than all 
the rest (it needs access to its JTextArea). So I handle that one 
specially; see the if statement inside the BufferPrimsTest 
constructor. The name parameter on the @Parameters annotation 
specifies which column to show as the name of the test. Omitting it leads 
to boring names such as 0, 1, 2, and so on.

I also test the line parsing code with a @Parameterized test. There are 
so many possible command input variations that it's not possible to test 
them exhaustively, but I think I have a reasonably good selection. The 
LineParserTest constructor takes a boolean (based on whether the 
line should parse successfully or not), the input string to try parsing, the 
expected line numbers, and any arguments such as the filename for a 
read command or the old and new text for a substitution.

You won't find anywhere in the test where I prove the functionality of 
JButton, JTextArea, and friends. These are both well-tested before 
release and also very widely used. You also won't find tests of simple
lambdas that bind action code to events in my SwingEditor.java file, for example:

```java
JButton goButton = new JButton("Go");
goButton.addActionListener(e -> doCommand(e));
```

This is trivial enough that I am fairly sure it won’t fail. The doCommand() method calls LineParser.parse(), which is extensively unit tested, and if all’s well, it calls the execute() method of the returned ParsedCommand. If doCommand() fails, no commands will work, and this would be discovered quite early. Even so, in the interest of completeness, there is a unit test in SwingEditorTest.java for doCommand().

I’m not a big fan of automated testing of UI code, but there may be times when you need it. There are several options for automated GUI testing, such as jimmy (which is currently maintained as part of the OpenJDK project) and Marathon, (which offers both free and commercial versions).

A good alternative is manual testing of just the GUI code. This consists of writing a very detailed test plan stating what the tester should do and what the tester should see.

```
Start the application, with a filename argument of 9
You should see a new window appear, containing text such as "Line 1", "Line 2", etc.
In the text box at the bottom, type the text *7d*, then press the Go button.
You should see line 7 disappear, so that "Line 6" is now followed by "Line 8".
Press the Go button again.
You should see the current line 7 disappear, so "Line 6" is now followed by "Line 9".
In the text box, select the existing text and hit Backspace or Delete.
The text box should be empty.
In the text box, enter the command *3,5s/Line/Slime.
Press the Go button.
You should see:
Line 1
Line 2
Slime 3
Slime 4
Slime 5
Line 6
...```

As you can see, this is very detailed and very tedious. But the alternative is a lot of tedious coding that will run fairly slowly because it will often re-create the GUI. So this is how I tested this application: manually.

**A Simpler Alternative Editor**

Given the complexity of vedj that I worked through above, what would have happened if I decided at the outset that the ability to use the powerful line-editing command mode from the line editor isn’t needed? In other words, what if I decided just a simple Notepad-style editor is needed? A simple graphical text editor can be built more simply, relying on the JTextArea to handle cut/copy/paste functionality for us. I described just such an editor in my earlier article, “The Command Pattern in Depth.”

You can still find the code on GitHub. It builds with Maven and there’s a run script for it. It has a menu bar, a graphical toolbar, a main panel, and no bottom panel. Like many design exercises, it is a bit sketchy but it works; you can load and save files, edit them onscreen using the mouse and keyboard shortcuts, and so on. But again, there’s very little design here; the work of maintaining the edit buffer is all done by Swing.

**Serial or Parallel Development?**
With this type of “dual UI” application—line and GUI editors—there is the question of whether to build one completely first and then build the other or to build them in parallel. The advantage of building one completely first—preferably the simpler one—is it makes manual testing of the second, more complicated version easier. In this example, I was in fact able to track down the source of some errors in the line mode of the GUI version by trying the same command in the line editor version. That told me whether there was a problem with parsing (which is common to both implementations) or with the GUI’s own version of buffer primitives.

On the other hand, building them both in parallel would have disclosed much earlier any limitations of the initial design. Fortunately, I encountered no major design problems in building the second version. I made only a few minor coding changes along the way, though these had to be reflected back into the line editor. For example, I changed the `Commands` class interface to allow the main program to provide implementations of a command letter, to facilitate the different ways of implementing the `a` (append) command. In the line-mode version, the program reads from the standard input; in the GUI version, it uses a `JOptionPane` dialog.

### Conclusion

How well did the original design fare in the transition? Quite well, actually. As mentioned, in `Commands.java` in the editor, I added the ability to add/replace a command letter and its command implementation. I smoothed out the interface a tiny bit. Overall, the design (which, in my previous article, I admitted to having adapted from the *Software Tools* books) survived pretty well in the transition from RatFor to C to Java and from a line-oriented editor to a GUI tool.

The editor described here offers a hybrid of line editing and simple screen editing. Thanks to some complexity in the code, you can mix and match modes; that is, you can select some lines of text via the mouse, cut them with a keyboard shortcut, and then undo this just by typing a `u` in the command area. I’ve explained many of the design rationales used along the way, in both parts of this series. I hope you find the information useful and an inspiration for those applications in which you build a specialized editing tool or simply want to customize the capabilities of a text area in your Java apps.