VM Plan of Attack

Felipe Bemfica and Isaac Jensen-Large

November 21th 2018

1 Project Breakdown

Vim exists as a relatively straightforward pipeline of commands from user input all the way to display on the screen. Key presses, inputted by the user, are transformed into commands, which are enacted upon the file currently held in memory. The file, having been changed, notifies a formatter about it, and the formatter requests what information it needs from the file and draws it onto the screen.

In order to organize this pipeline in a clear way, we have separated it into three responsibilities:

- The Model is responsible for storing all of the data currently in a Buffer (the file being edited, as held in memory), as well as information about the state of the editor, such as register contents.
- The View is a complex interface for drawing on the screen, and is responsible for showing the UI to the user, as well as for receiving the user's input into the UI.
- The ViewModel is responsible for maintaining parity between the View and the Model. As the user inputs commands into the View, the View forwards them to the ViewModel, which parses them into operations to perform upon the Model, and then in turn formats the relevant information from the Model and supplies it to the View, so that the View can display changes to the user.

The ViewModel has complex responsibilites, so we've subdivided it into two major modules:

- The CmdGenerator is responsible for processing raw input from the View and turning it into Transactions that can be performed upon the Model.
- The Formatter is responsible for retrieving data from the Model when it is notified of change, and for formatting this data so that it can be displayed through the View.

1.1 The Model

The Model is responsible for the current Buffer and for the non-Buffer-specific data that is still relevant to the state of the program, most notably the registers. The Model stores its Buffer as a component, in order to make it more extensible, and make multiple-Buffer editing easier to implement at a later time.

The Buffer itself stores its data as a doubly-linked list of linked lists of strings, where each linked list represents a single line. This unconventional data type is meant to increase the viability of editing large files; a doubly-linked list allows the editor to access the lines close to the cursor without having to count all the way up from zero, while still making it efficient to insert lines. Each line, in turn, is represented as a linked list of strings so that files which are formatted as one long line can still be inserted into efficiently, without sacrificing indexing speed to too great an extent.

1.2 The View

The View is comparatively simple. It draws text to the screen and receives input from the user using ncurses, and exists to provide interfaces so that the ViewModel can interact with ncurses at arm's length. What is actually done with the inputted raw characters or what is actually drawn to the screen are the responsibility of the ViewModel.

1.3 The CmdGenerator

The CmdGenerator parses user input and creates Transactions based on it. The parsing will be done by a finite state machine, which will resolve a decision tree based on raw user input to find out which type of Transaction it should instantiate. It will then instantiate a transaction and execute it on the appropriate Buffer, as provided by an interface to the Model.

1.4 Transactions

A Transaction is a small object which is provided interfaces to a Buffer and to the Model, and which has a method called execute(), which performs the transaction upon the given Buffer and model. Transactions represent simple commands like "delete until the end of the line," or "delete the current word and change into insert mode," or "yank ten characters to the right of the cursor into register a," representing `d\$`, `cw`, and `"a10yl` respectively.

In order to create these complex commands, we will implement Transactions with the Decorator pattern, such that something like `10d` is a Multiplier Transaction which owns a Delete Transaction.

1.5 The Formatter

The Formatter is an observer that is watching a Buffer. When that Buffer is modified, its Formatter is notified, and the Formatter makes queries to the Buffer in order to get the text that it needs, and executes commands upon the View to draw the relevant parts of the buffer to the screen.

2 Design Plan

First, Felipe will begin writing the View and Isaac will begin writing the Model. Both of these work well in isolation, as they don't need to handle other objects, they just provide interfaces to be handled. The Model will probably be a lot of work compared to the View; Felipe should be able to finish the View within a few days, and will move on to begin work on the Formatter. The interface between the Model and the Formatter is relatively abstract, so that should be writeable even as Model hasn't been finished yet. Formatter should only require another few days to write.

It is difficult to predict how long Model will take to write, because of its complexity and because of our unpredictable schedules leading up to exams. So, while we can provide a fair timeframe on View and Formatter, we will have to anticipate a range of possible durations for the development of Model. With this in mind, when Felipe is finished with Formatter, he will start writing CmdGenerator. When Isaac finishes the Model, he can help write the Transactions; Felipe may have already begun to write these if he's finished CmdGenerator by this point.

By now, we are probably approximately two weeks into our three-week development cycle. We are finishing up the remaining Transactions, and implementing additional features such as Syntax Highlighting and Macros. After cleaning up any loose ends, the project is complete.

3 Questions

3.1 What would it take to support multiple files at once?

To support multiple Buffers, the Model would hold references to more than one Buffer. The CmdGenerator would be aware of which Buffer was currently being edited, and would supply each Transaction in turn with an interface to the appropriate Buffer. Each Buffer, upon being modified, would notify its respective Formatter, which would draw the new version of the file to the screen. If the active Buffer were changed, the CmdGenerator would start feeding a different Buffer interface to its transactions, modifying a different Buffer and causing a different Formatter to draw different data to the screen. Simply displaying a different file would be done by calling a "wake up" method on the Buffer, which notifies the observing Formatter but does not exact changes on the Buffer.

3.2 How would we notify the user that the file the user is trying to modify is read-only?

Suppose we wanted to warn the user, but allow the user to edit the document while preventing them from saving. Information about the permissions of the file in question would be stored in the Buffer. Then, transactions which mutate the file would inhereit from a parent which warns the user upon trying to edit a read-only file, and then flips a bit in the Buffer so as not to warn the user a second time. Upon trying to save, the special transaction which would perform this would notice the read-only status of the file and would refeuse to save, throwing another alert at the user.

Conversely, if we wanted to prevent users from editing a read-only file at all, we would similarly make Transactions which mutate the Buffer inhereit from a parent Transaction which would refuse to act on a file with a read-only bit.