Plan of Attack:

Overview:

Firstly, we must consider the features our editor must be able to handle. With a brief overview of the requirement, whatever “thing” we must wrap our document in must be able to handle insertion, deletion on a line to line basis, and all while keeping track of the words in each line and their states. My approach to this would be to abstract the model, controller, and view. With the concrete Vm class extending from the model, ncursesView extending from the view, and keyboard control extending from the controller. On a state level of the program, I plan to keep a few enumeration classes to mark the action inputted by the user and the possible highlight the “word” should appear in. furthermore, the Vm class will contain a history vector of Action that will keep track of the commands that can be undone and executes these when the appropriate action is triggered.

File Abstraction:

As mentioned above, I plan to abstract the file read designated by the user into a File structure that keeps a vector of Line structures. The reason I am doing this is that while we can read in the file content line by line as given, the window we have open does NOT guarantee support for that line dimension. Thus, I believe it is needed to create a File class that offers seamless conversion from “raw” chars to Line objects, respectively to allow the best user input and word interpretation for syntax highlight. Specifically, for syntax highlighting, I plan to have each word contain a field called color that is an enumeration. So when a word is created, it would be searched against a library of keywords and decide its colors.

The order of support for commands I plan to implement in order can be generally mapped out by first displaying the contents, permitting cursor movement, writing content(including commands like :wq and such), word abstraction, word insertion or deletion, and lastly things like copying, joining, and pasting.

NcurseView:

For the view terminal, I plan to have it completely implemented in the class NcurseView that extends from the general view superclass. The idea is basically to have the NcursveView class "have" a Vm object ptr, thus is able to make direct changes to the vm content and later save it after taking an action is triggered from the keyboard class. Generally, the NcursveView class must take full control and restriction of the user's cursor movement and displaying the Vm content. I plan to do so via the ncurses library and abstracting some parts to suit this project.

Question: Although this project does not require you to support having more than one file open at once, and to be able to switch back and forth between them, what would it take to support this feature? If you had to support it, how would this requirement impact your design?

I think to support multiple file opening, we would consider having a 1 …..n has a relationship from Vm to the File abstraction class, thus the Vm class would keep a vector of File objects. Since I have abstracted my file format, this wouldn't impact too much, aside from the need to

perhaps abstracting the history field to a vector of vector of actions, where each one would respectively correspond to a File object.

Question: If a document's write permission bit is not set, a program cannot modify it. If you open a read-only file in vim, we could imagine two options: either edits to the file are not allowed, or they are allowed (with a warning), but saves are not allowed unless you save with a new filename. What would it take to support either or both of these behaviors?

I think in this case, we would want to first determine if it's possible to save and warn the user if it's read-only. With this field determined in Vm, we would want to have overloaded versions of writing to file that takes a new name as a parameter to be enabled appropriately. If we want to implement support for save, we can enable an overloaded function to save. And if not, we can disable both save functions and displaying such status on the screen in ncursveView.

Timeline for Completion:

1. Displaying of file completed (Nov 28-29th)
2. Cursor movement completed (Nov 30th)
3. Writing to file and confirming the combination of existing commands are compatible(Dec 1st)
4. Implement word related functions such as a move to next word, replace the word, delete the word, and so on (Dec 5th)
5. syntax highlighting (Dec 6th)
6. add any miscellaneous commands like copying, pasting, and joining (Dec 10th)
7. Lastly, general debugging and testing (until the deadline)