Executive Summary

The main type of architecture chosen to implement our text editor is the MVC architecture. Due to the relative simplicity of vim, the model is the most straight forward of the three to implement, with display coming in second and the controller being the greatest headache for the implementor. This report is structured into four sections, three of which tackles our plan of attack for the Model, View, and Controller, with the fourth adding extra information as deemed fit. Details covered include predicted challenges, proposed solutions, limitations of planned implementation, and how applications of good design principles leave room for extensions beyond the scope of the requirements. Complete implementation details may be skimmed over as the UML does a wonderful job of picturing the links between our program and higher-level groupings of responsibility will be explained instead.

Model

The model simply serves as a buffer between the user's edits and the actual process of writing to files. The data structure would load the contents of each file into memory upon opening a file and be continually updated through user edits. Data structures considered include a vector of strings representing lines of text, a doubly linked list of doubly linked list of characters, and the rope. After weighing the insertion, removal, and access times for each of the three choices, the group decided that insertion and deletion happen more often than arbitrary retrieval, favouring our choice for the nested doubly linked list.

As is typical for the MVC architecture, the Controller would react to user input and update the Model through its public interface. Changes to the buffer will be communicated to the View wide through an abridged version of the Observer Pattern which is explained in detail in the following sections.

To represent the cursor position, the file buffer "owns-a" PtrCursor object, which internally uses iterators to represent the position of the cursor in the buffer, reducing the need to linearly traverse our model through every sequential insertion.

Upon opening a read-only file, we can set a state bit in the corresponding file buffer which is verified upon attempting to execute a MutateCommand (see Controller section). On the other hand, should we allow edits to the buffer, we may simply throw an exception when the user attempts to execute a :w command on the same file. Neither solutions are too difficult to implement but the second one is more aligned with how vim handles this case, thus this is the way we shall handle it.

View

As aforementioned, the View-Model exhibits the rough shape of the Observer Pattern. Each file buffer may have several abstract BufferViews attached once which lends nicely to potential extensions to a graphical user interface and splitting screens, features of the actual vim editor.

In our specific implementation of the BufferView, we were inspired by both the Adapter Pattern and the popular ReactJS library. We wrap the C interface of the NCurses library through a rough approximation of the Decorator Pattern in which each Component will be responsible for display a single thing, such as the status bar, the actual text file, or the borders, and composing Components upon each other to form the whole terminal interface. Should we choose to implement it, the accumulation of these ideas naturally forms a basis for syntax highlighting and other decorative display features as these can be composed on top of a text display component.

Our main issue would be familiarizing ourselves with the NCurses library and bridging the gap between the C library and our C++ project.

Building on the potential target of syntax highlighting, in order to handle contextual keywords, we may employ the standard regex library or even adjust our solution to assignment 3 to verify if a line contains an instance of contextual keywords. This method could also be used to match for literals on all of the other cases necessary for syntax highlighting.

To support vertical or horizontal splitting, we can conceptualize that our Controller has a map of models from which it may switch back and forth. The active screen would be the one which is currently active from the Controller. The rendering and resizing should be the complete responsibility of the View class and should be tricky but a straightforward calculation.

Controller

By far, vim's rich set of commands and ability to aggregate different commands is its biggest selling point. However, it is also the biggest headache for the implementor. Consider the following example: "3c3fA". This sequence of characters, when pressed in the default command mode, would cut 3 times, from the current cursor to the 3rd occurrence of the character 'A' on the line. However, somewhat counterintuitively, the entire length of the cut over the 3 iterations would be placed onto the buffer and not just the last cut command. Even more of an issue is the undo command. As almost every command which mutates the state of the file buffer is undoable, it suggests the necessity of a general solution to the desired functionality of commands.

After much consideration and toying with ideas such as a map of lambdas. The group has settled on a massive inheritance scheme starting from the base class Command. In its different modes, the Controller would be responsible to parsing, interpreting, and creating the appropriate subclass of Command which overrides a doCommand() function, not unlike the solution to regular expressions in assignment 3. Our parsing would simply be greedy though as we need only run the first valid command typed by the user, greatly easing our burden.

To handle the myriad of different categories of commands, we employ the Interface Segregation Principle and Multiple Inheritance to have each concrete Command sub-class inherit from different abstract sub-classes of Command. (A command may be one, two, all, or none of, Undoable, Move, and Mutate Commands).

In order to support undo and redo support, we hold two stacks of UndoableCommand in the controller, popping and pushing into each of them as necessary. When an UndoableCommand is created, a pointer is pushed onto the undo stack. Each UndoableCommand overrides the redoCommand() and undoCommad() functions, which are called as appropriately when the user asks to undo or redo their commands. Note that this solution would enable unbounded undo as desired.

Given a working implementation of the Command hierarchy as described above, it should be a natural extension to support macros. We would simply registers containing macros in the form of sequences of Commands. Complications may arise in the relative and absolute positions where

we execute a command but it may be solved by storing the current cursor state upon calling doCommand().

To handle input, we abide by the Liskov Substitution Principle and have the Controller own an abstract Input class. For this specific instance, we would simply use the NCurses input function but clearly, this can be generalized to use other libraries.

Schedule

1. Design
    - done
2. Interface
    - done
3. Basic Functionality
    - Saturday Nov 24, 2018
    - Working basic editor with minimal set of commands and no undo
4. Extend Functionality and Bonus
    - Saturday Dec 1, 2018
    - Full set of commands, all modes, undo, redo, syntax highlighting, and potentially macros

Delegations of Responsibilities (Basic Functionality)

- Felix
    a. Commands
    b. Controller
- Peter
    a. Model (file buffer)
    b. View

Further delegation of tasks beyond the basic functionality is yet to be decided.