Overview

My implementation of Vm takes advantage of the Command Pattern. In this pattern, each Command is a concrete class of the Abstract_Command class which has an `execute()` and an `undo()` method. An Invoker gets passed concrete instances of Abstract_Command and blindly calls these methods.

Design

main.cc

The main.cc file is quite short -- only 37 lines -- and only does a few things. It checks if the program was executed with a file path, and it notifies the user if it wasn't. If a file path was provided then it instantiates the Editor, NCurses_Display, and Controller, and calls Editor's `init()` method on the latter two. Finally it calls Editor's `run()` method to actually start Vm.

Editor

Holding the entire implementation together is the Editor class. This class holds the File, Page, Cursor, Clipboard, and Invoker objects, as well as pointers to the NCurses_Display and Controller. To minimize coupling, if any of Vm's objects need to access each other, it does so through the Editor's corresponding getter method. When the Editor's `run()` method is called, there is an indefinite while loop that runs which continuously takes in Commands from the Controller, passes it to the Invoker which executes it, and then refreshes the display. As long as the Editor's `running` field is **true**, Vm is running. Inside the while loop, taking in Commands is encapsulated by a try-catch pair which attempts to take Commands until a valid one is returned. This way, if an invalid input is provided to the Controller, it can just throw without crashing the program.

File

The text in the real file is read in line by line into a vector of strings which is stored in the File object. This vector of strings (with field name `the_file`) is what is being edited on throughout the execution of the program. When this working file is ready to be saved into a real file, we first check that Vm has permission to write to the real file. Once that's confirmed, only then do we erase all of its prior content and then write the working file's content to it.

Since `the_file` is a private field, if a Command needs to edit the file, it must do so via File's public methods. These methods have built-in checking to make sure that illegal operations are

not called. For example, if a Command tries to delete a line that does not exist, File's public method that the Command called will stop this. This reduces the possibility of segmentation faults and other deadly bugs.

Page

Although the entire file is stored in the File, we only ever show a small sample of the file to the user at one time through the screen. This means that I should have an abstraction that the view can call on to get all the needed information which needs to be displayed. This Page class keeps track of which lines of the text should currently be shown and returns that back to the view. It also keeps track of which location the cursor should be relative to these lines.

NCurses_Display

The NCurses_Display class is how Vm displays all the relevant information to the user. When its public `refresh_display()` method is called, it calls the Page's getter method to receive and print all the text that needs to be displayed to the user. It also does the appropriate calculations using the screen size to have the text-wrapping feature. The information in the status bar is collected through simple calls back to the Editor's getter methods.

Controller

The Controller is responsible for taking in input from the user. Since Commands are determined by which input the user gives, I have also made the Controller responsible for passing Commands back to the Editor.

One challenge that I faced was trying to separate the user's input into the multiplier and the actual command. I ended up using the built in `std::stoi` function to continuously check whether the input provided thus far was an integer. Since `std::stoi` throws if it is not an integer, I took advantage of try-catch blocks to mark the end of the multiplier substring and the start of the command substring.

If an input string does not correspond to a Command, it clears the string and retries. However, for Commands that are parameterized by another Command (e.g. `d[ANY MOTION]`), if the second input is not a valid Move_Command, the Controller throws instead.

Once the correct Command to instantiate has been determined, the Controller then creates a unique_ptr of this Command with its multiplier and returns it back to the caller.

Invoker

The Invoker is what actually executes the Commands. When the Editor calls the Invoker's `invoke_cur_command()` method, the Invoker calls the `execute()` method of the Command the number of times corresponding to that Command's multiplier. The Invoker doesn't know what Command it is invoking, only that it is a subclass of Abstract_Command.

After the current Command has been invoked, the Invoker moves the Command to the `history` of commands, implemented as a vector<Abstract_Command>. The reason why we need a vector and can't just store the most recently executed Command is because not all Commands can be undone. When the last Command needs to be undone, the Invoker loops backward through the history, calling each Command's `undo()` method and popping that command from the vector until it has done so for a Command that can actually be undone.

Commands

At the top of the class hierarchy is Abstract_Command. The Editor and Invoker only know of this class's existence. This means that every other Command must be a subclass of Abstract_Command and do its job independently when its `execute()` or `undo()` methods are called.

Since a Command needs access to any combination of the File, Cursor, Controller, or Display, it must receive these by calling the Editor's getters when it is executed.

For a Command that doesn't get undone, its `undo()` method body just returns **false** to tell the Invoker that nothing was undone.

Move Commands

Most Move_Commands work by getting the initial Cursor position from the Editor, doing the appropriate calculations to get the final Cursor position, checking that this final position is in bounds (and bringing it in bounds if it is not) then setting the Cursor position through the Editor. For the Move_Commands that are context dependent (e.g. ones that move to the next word) they also ask for the File from the Editor.

Move Commands cannot be undone, so their `undo()` methods just return **false**.

Edit Commands

Some Edit Commands (e.g. `i`, `a`) put Vm into insert mode. When such Commands are executed, they call a common `engage_insert_mode()` method in the Abstract_Edit_Command

superclass. This mode continuously asks for input from the controller and then calls the appropriate method from File to do the edit. The same happens for Commands that put Vm into replace mode.

Since Edit_Commands make a change to the text stored in the File, they can be undone. In order to support undoing, we keep a vector of tuples in Abstract_Edit_Command which details how the file was changed and in which order. Each tuple records that character X was INSERTED/DELETED at position Y. And the tuples are stored in chronological order in the vector according to when the edit was made. When the `undo()` method is called, the vector is looped through backwards and the opposite DELETION/INSERTION occurs to restore the File back to the state that it was in prior to the execution of the Edit_Command.

When a `\n` character is inserted or deleted, a corresponding line is created or destroyed in the File.

Other Commands

Since we have the Page class, the Commands which move the view window up or down the page can simply make a call to it.

The copy Command just takes whatever is currently on the line of the Cursor and saves it in a Clip object which is then passed to the Editor's Clip setter method.

The Colon Commands are quite straightforward. Saving and quitting can be done with simple calls to File's and Editor's methods. `:0` and `:$` can both just instantiate and execute corresponding Move Commands.

Extra Credit Features

I implemented real-time window resizing. In my Vm, you can resize the screen at any time and the program appropriately shows more or less text. It also recalculates which lines need to be text-wrapped on the fly. I was able to do this because NCurses has a special character that is returned by `getch()` if the window was resized. So in my `get_input()` method, Vm continuously refreshes the display while this character is being returned by `getch()`.

Final Question

Building this project was one hell of an amazing experience. I'm extremely glad that I discovered the Command Pattern in the design stage of this project. The abstraction that it provided allowed for each of the Command's jobs to be compartmentalized. It also allowed me

to easily add more Commands one after another. If I had to start this project over, I would have tried to implement syntax highlighting and Macros earlier, rather than focusing on implementing as many Commands as possible. I feel like Macros shouldn't be that hard with the abstraction over Commands that we have. It seems to me that Macros can just be a subclass of Abstract_Command. And each Macro can just hold a vector of Abstract_Commands. When the Macros are executed it can just call `execute()` on every Command that it holds in order. Similarly, when the Macro is undone, it can just call `undo()` on every Command backwards. I haven't really thought of a good way to do syntax highlighting.

What a great way to end off the by far the best course I've taken at UWaterloo so far.