VM Final Design Document

F24 CS246E Final Project, VM (Vim) Final Design Document + UML Diagram

By: Patrick Lin

Overview

This is the VM (Vim) project. I chose to support macros (syntax highlighting is not supported). This program employs MVC architecture, among other object oriented design principles. Memory safety in this program is ensured by wrapping pointers in unique_ptr objects owned by other objects that are guaranteed to last the duration of the program. Key features include a highly modular design for commands, which allows for easy future extensions.

Model

The model abstract class (and its owned classes) represents the state of the editor. They contain the actual data. For example, VMState is the concrete subclass of Model, which owns several other "state" classes like FileState, CommandBarState, Clipboard, EditHistory, and Macros. Each of them is responsible for a specific area of data.

FileState manages the file itself, including the file contents. It stores a copy of the file in memory, and manipulates the contents directly through abstractions like insert, replace, remove, etc. It also manages the IO to the file. It maintains cursor position (as opposed to managed by the View), and provides some abstractions like setCursor and a safer moveCursor (which guarantees a correct cursor position).

CommandBarState manages data related to the command bar, such as the actual contents of the bar. It provides several abstractions for fine grained control demanded by several features (for example, recording macros requires the recording message to be showing for the entire duration). It also manages data for search functionality, which must persist for the duration of the session. Actual search implementation is done in the Actions.

Clipboard is very straightforward and manages one string, the clipboard contents.

EditHistory manages the edit history of the file. It also keeps a local cache of the file content, initialized on construction by passing in the FileState. It uses this copy to diff the file and create a Change when createChange is called. These changes are then stored in a stack. To implement redo, VMEditHistory also stores a `recentEditCmd`, which is basically the command issued by

the user to create the most recent change. When redo is called, `recentEditCmd` is passed into the Controller as a replay (which is the same channel used by Macros) so the exact same edit can be carried out again.

Macros stores the macros created by the user. It maintains an unordered map of char to int vectors (which are character buffers). When a record is started, the `currentRecording` char is set to the macro, which is then used by VMState to display the "recording @" message, as well as Controller to send new keypresses into Macros via `append`. Then, when replay is called by an Action, Macros sends back the corresponding character buffer back into the Controller as a replay, which processes this new character buffer as though they were user keypresses.

View

The View abstract class represents the visual aspects of the editor. They (the concrete subclasses) directly interact with ncurses and provision, manage, and write to the windows they correspond to.

They each also override a `displayView` function, which rerenders the window and hence updates the contents. The original plan was to optimize this rerender and only update the necessary parts. However, due to time constraints, I opted to do a full rerender.

Each of these views also has a pointer to their corresponding state (FileView to FileState, CommandBarView to CommandBarState). The views each call the state's respective data fetch methods to write contents to the screen.

Model owns the views, which are initialized in VMState and added via `addView()`

Controller

The controller abstract class represents the user interaction, i.e. keypresses. Its only concrete subclass, CursesKeyboard, interacts with the ncurses library to get keypresses from the terminal. It also provides mechanism for replaying user input, which is described in Macros. It maintains a pointer to Macro, which it calls for append when Macro is in recording mode. It also maintains a persistent buffer for insertion/replace actions, to get the overall changes when inserting/replacing. Finally, it manages a multiplier if a prefixed-multiplier is input by the user. For example, `2j` will place 2 into the multiplier, since it occurs before the buffer is populated with `j`.

On each keypress, CursesKeyboard queries VMInputParser for an Action pointer with the current buffer (which may not necessarily succeed, in which case a nullptr is returned), which is returned to the VMState for execution.

VMInputParser is a class owned by CursesKeyboard. It has a list of Action objects (which VMInputParser owns), and are manually entered into the list by importing the corresponding Action in `parser.cc`. This VMInputParser is responsible for returning a pointer to one of the Actions given an input buffer.

Action

This is where the bulk of the functionality of VM lives. The Action abstract class represents an action to perform. For example, moving a cursor, or inserting text, or searching, etc. Each Action has 2 main functions: `doAction` and `matchAction`. `matchAction` takes an input buffer, and returns a boolean based on the contents to indicate whether or not the input matches one of the functionalities the Action implements. `doAction` takes the input buffer and performs the functionality on VMState.

Each Action has 3 properties that can be queried via their getter functions: `mode`, `redoable`, and `selfdefmultiply`. `mode` is chiefly used by the VMInputParser to match the action - it can only be matched if the Controller has the same mode as the action. `redoable` was the original intended method of implementing redo. However, it has been deprecated in favour of extracting the `textentrybuffer` and replaying it using the Macro mechanism. `selfdefmultiply` indicates whether or not the Action defines multiplier functionality on its own. If it does, then the `doAction` caller will supply another parameter, the multiplier. If not, then the caller simply calls `doAction` without the multiplier parameter, and calls it `multiplier` times (in a loop).

Each of these actions receives a `VMState` pointer in their `doAction` call, so they have free reign over manipulation of the VM editor using the provided public methods of each class. Some of these Actions also call other Actions. For example, CopyPaste calls MoveCursor when yank/delete/change is called with `[any motion]`. For this purpose, CopyPaste will own a MoveCursor via a unique_ptr. This is acceptable because the Actions are VMState-agnostic and don't store any data, and can be considered static.

Because each `Action` mostly lives independently from one another, this allows for a highly modular design: a new command can be supported by creating an `Action` class and specifying it in `parser.cc`

All Together

Together, these classes make up the VM project. When running, a `main.cc` simply instantiates a `VMState` as a `Model` unique_ptr. Then in a loop, while the `VMState` 's exit flag is not set to true, the `run` function is called, which does the following:

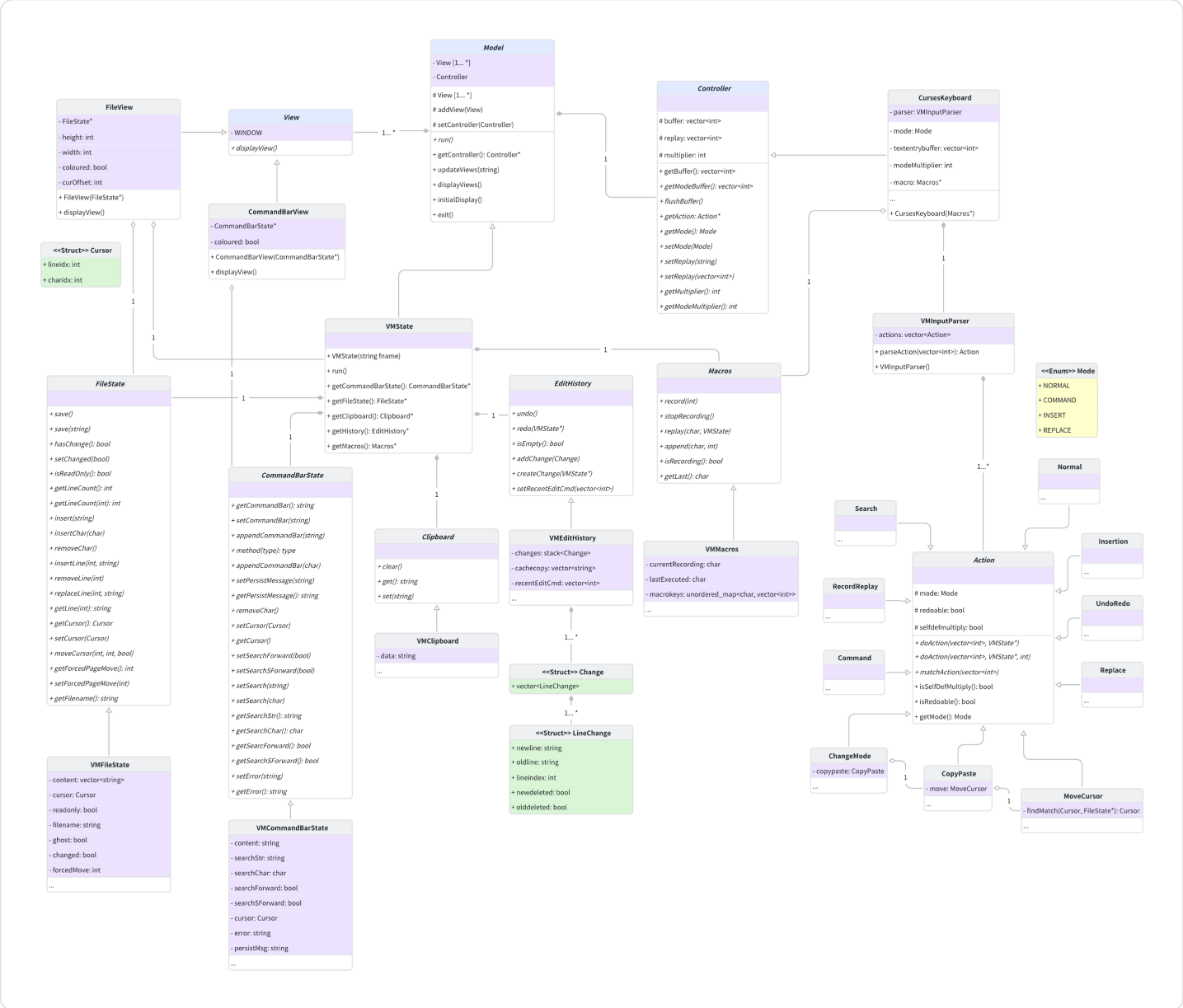
1. Fetch Action from Controller (stalls on user input)
2. If Action is not a nullptr, call it
3. (Re)render the views

This design is inspired by the fetch-decode-execute cycle seen in CPUs.

UML

This UML has been updated from the original design. Several key changes include adding a CommandBarView class to maintain a clear separation between FileView, and making the Controller responsible for updating the Macros model. Due to the nature of multipliers and the `[any motion]` commands, I also made ChangeMode own a copy of CopyPaste, and CopyPaste own a copy of MoveCursor. The remaining classes are largely unchanged, except for adding more methods for better abstraction.

(also submitted separately as dd2-uml.pdf)



Design

File Manipulation

For the purpose of this project, I elected to go with a simple `vector<string>` to store my file contents. Although I originally researched gap buffers for its superior performance, it was scrapped to meet the time constraint. Hence, primitive file manipulation features like adding lines, removing lines etc directly call the corresponding vector functions `insert`, `erase`, etc. More granular functions like `insert(char)`, `removeChar()` use the current cursor position to call `insert`, `erase` on the string stored in the vector. Although this is not efficient, it was quick and painless to implement.

File IO is very straightforward and uses C++'s `<fstream>`. Errors on fstream are handled gracefully. To determine if a file is read only, I decided to first open an `ofstream` to the file in append mode. If it succeeds, then the file is not read only, which is much cleaner than

navigating the platform-dependent mess that comes with attempting to read the permission bits of the file. If a file is readonly, writing is not permitted unless `:w` is supplied with a different filename.

Finally, a `changed` flag in the file state indicates whether or not the file has been modified. This flag is set whenever any of the insertion/remove/replace functions are called on the file state. The flag is unset whenever the `EditHistory` stack is popped to empty, allowing exit through `:q` when `u` is held to the end.

I also decided to support opening VM without specifying a file (aka blank state). In this case, `:w` must be called with a filename argument.

Edit History

The edit history of the file is preserved in the following way: first, EditHistory is initialized with a copy of the file contents. Then, whenever an action modifies the file contents, it will call EditHistory's `createChange` function. This function takes in the VMState pointer as a parameter, and diff's the contents of the current file against what is cached. This difference is stored as a Change object, which is a list of LineChanges, each containing the old line and the new line. Then, when undo is pressed, the `undo` function iterates through the changes in `Change` and replaces the corresponding line with the old line contents.

Originally, the redo command was intended to take the last change and analyze the difference to repeat the change. However, this was near impossible to do, so instead I opted to store the characters the user entered to initiate the insertion/replacement, and also the characters entered during that process. These characters are reported by the Controller, and redo then uses the Macro replay mechanism to redo the change.

Clipboard

This was very straightforward. The clipboard stores a string.

The main challenge was getting `[any motion]` to work. In the end, I decided to take a substring of the input, and feed it into `MoveCursor` to invoke as a sub-action. Then, the difference in cursor position is analyzed, and the result is taken for copying/deleting.

Macros

To solve the challenge of macros, I decided to have the controller maintain a pointer to the Macros state. If the Macro is currently recording, then the Controller will report the entered characters to Macro in addition to the current process. Then, when the Macro `@` command is

invoked, the characters are then sent to the `replay` buffer of the Controller. This `replay` buffer is taken as a priority before user input. The Controller treats incoming characters from the replay buffer as though it was user input, and matches actions and runs accordingly. This also means during the replay, the fetch-execute-display cycle of VMState doesn't stall, and repeats until the `replay` buffer is emptied.

Commands

Normal commands are straightforward and can be understood in the above Overview section.

Colon commands are a bit special in the sense that the command is not formally "issued" until the user presses the enter key. Additionally, the user can backspace to rewrite the command before entering it. To implement this, I made CommandBarState implement a subset of FileState's content manipulation abstractions, targeting a single line instead. Then, whenever `:` is issued, the Controller enters the `COMMAND` state, at which point every single keypress matches with the `Command` action. This action handles rewriting/entering. The input buffer is not cleared until the enter key is issued and the command is processed.

Searching

Searching is another special command that is not issued until entered, and can be rewritten. To keep my sanity, I merged `/` and `?` Into the same action as `:`, and they also switch to `COMMAND` mode. When entered, they store the search content into CommandBarStare. Then they abuse the Macro replay functionality to call `n` for a search in the same direction. It wasn't until later that I realized a Piazza post revealed we had to include `/` and `?` as part of `[any motion]`. Due to this design choice, I unfortunately did not have time to rewrite it as part of `motion`.

Extra Credit Features

N/A

Final Question(s)

- What would you have done differently if you had the chance to restart?

I definitely would have started with a much deeper dive into Vim than what I did. Because I was an occasional Vim enjoyer, I thought I had a decent grasp on the functionality I needed to implement. However, I neglected the multiplier support in my initial design (among some

others), which led to some suboptimal workarounds to make them work. I also should have begun way earlier... I underestimated the amount of work needed and, especially since this is an exam period, my mental and physical health deteriorated very quickly. Lastly, I would invest way more time into the design phase. I made some poor design choices that required significant efforts to correct, which would've not happened had I spent more time designing in the first place.

- 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?

To support this feature, we can abstract VMState's contents into a `Context` class. The new `Context` class is the model that owns the views, controllers, states, etc. `VMState` remains as the "gateway" model class, and provides a mechanism to switch contexts such that the accessor methods that return views/controller/states return the current Context's values. Overall, this is a pretty minor change in design.

- 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 behaviours?

My current design implements the latter option: saves are allowed only if it is a new filename. To prevent platform dependent issues with checking the write permission bit (since I am developing cross platform), I decided to check the write permissions by attempting to write and checking the result. This check is done when the program is run and the FileState is instantiated. To implement the first option, we need to modify our actions so that edits to FileState are not allowed when the file is read only. Since the design is so uncoupled, this requires a number of changes to both the editing Actions as well as VMState to check the readonly flag first before editing.

Conclusion

I'm happy I finished (mostly)! I think, in retrospect, this is a very cool project :)