CS246E VM Documentation

Padam Chopra (p5chopra)

15 December 2020

1 Introduction

The final project for the Fall 2020 offering of CS246E was to make a vim-like text-editor called VM. The vm text-editor was to have similar features as that of Vim, that is, being able to switch between modes like Insert Mode and Command mode to edit and save files with easy key shortcuts to execute actions.

The class was divided into pairs and each member of a pair had to implement either Syntax Highlighting, or Macros. After discussion with my partner, it was decided that *I would be implementing Macros* and he would work on Syntax Highlighting. My first experience with *"programming"* was making Macros in Excel so I thought this would be an interesting thing to try to implement.

The project was meant to test us on what we have learned about the Object-oriented programming paradigm throughout the term and to come up with a design for VM that could easily be modified to support changes in the future if any of the specifications were to change or if additional features were introduced.

I have employed the Model-View Controller (MVC) architecture pattern to structure and design my implementation of VM. Therefore, the operating logic, interaction with user inputs, and displaying output to the user are all segregated to *minimize coupling.* More discussion on the architecture and how it makes achieving our objectives with the project easier is described further ahead in the document.

On top of the MVC architecture, my design employs multiple techniques like *class hierarchies, overloading, and container classes* to accomplish a lot of the objectives that we had with the project. I make use of the *NCurses* library to make my program interact with the terminal, and later on this document, I will also discuss what would happen if this library was to be updated with new interfaces or if we had to switch to a different library.

To run the project, you just need to navigate to the project directory and make use of the *make* command. This creates an executable file called *vm* that you can open with fileNames as arguments.

2 Overview

The overall structure of my implementation is easy to follow. Let us start by looking at the three main components of my design.

2.1 Model

The model contains all the logic needed by VM. It has a base *Model* class that has essential methods like updating views, and getting input sequences from the controller. Thus, it also owns a *Controller* and a vector of *View* objects to be able to do so. As of now, there in only need for one concrete Model, that is, *VMModel* and it inherits from this *Model* class. VMModel holds states, called *WindowStates*, for each file buffer that the program was opened with. It also holds a list of the *Modes* that currently exist in VM:

- Insert Mode: To start typing directly into the current file buffer on the screen.
- Command Mode: Arguably the most powerful mode in VM. Used to issue commands to ease editing, navigation, etc.
- Ex Mode: This is the "admin" mode, as it interacts with the system and can be used to save, navigate files as well as quit VM.

Both Command Mode and Ex Mode own a list of commands that can be executed when the user is in either of those Modes, and they are discussed more in-depth in the next section.

2.2 View

View contains all the logic necessary to display information on the screen based on the *WindowState* that it is provided with. The base class *View* has essential methods like *update* and *updateStatus* that the model can call to notify the view to refresh. My implementation only requires one view i.e. the terminal/shell so my *TerminalView* class inherits from the *View* class and is used throughout the lifetime of the VM.

2.3 Controller

Controller is responsible for interacting with the user and getting their input by recording their keypressed and sending them to the model. To do so, it makes use of a method called *getInput* and sends a sequence of character codes back to the model. It also helps clear recorded inputs whenever the model asks for it.

3 UML

A UML diagram has been submitted along with this documentation that contains information about the design of my implementation of VM. Although there are major chunks that are the same as what my UML at Due Date 1 mentioned, there are some changes that I would like to walk-through.

3.1 MVC super-classes

Initially, I planned on having a *VMModel*, *TerminalView* and *TerminalController* class directly. While building VM, I realised that there might be possibilities in the future where we have a different kind of View- say a GUI view, or maybe two shells instead of one. Extensibility would be really difficult if I had only one concrete view class. A similar case could be made for controller, and model. That is when I decided to make super-classes for each of them and have virtual basic functions that could be overridden in case implementation was to differ in any way.

3.2 FileManager

My *VMModel* and *MacroManager*, both had a *"has-a"* relationship with my *FileManager*. After we were introduced to static methods, and based on my experience with OOP, I decided to implement static methods within *FileManager* as they did not really need an instance of the class to function and were more accessible as static methods.

3.3 StatusBar

I changed my decision of making a *TerminalView* own-a *StatusBar* into the base *View* class having a method to update statuses. This also makes it easier to overload the implementation by Views and do it differently than how the base class does it.

3.4 CommandType classes

In the UML, ⟨⟨ *CommandType* ⟩⟩ stack of classes that inherit from Command is a completely new addition to my implementation of VM. I noticed that in commands with the structure *command [any motion]*, it was important to be able to differentiate between the kind of commands. It was also easier with this segregation to change properties of commands in groups. For example- all of the motion commands were modified to not be *undoable*. It also aided in implementation of some commands like . that operates differently if last command run was of type *InsertCommand* than when it was of another type.

3.5 ExMode

ExMode is a completely new mode that did not need to be created according to my plan as of Due Date 1. It was created out of need to align with our primary objective of easier extensibility in the future. *ExMode* commands work differently than *NormalMode* commands as they are not countable, and can often be run with a parameter. They are also shown on the screen as the user types them, and therefore, to accommodate for these many changes and to allow more *ExMode* commands to be added in the future, this was implemented as a separate mode that owns a list of *ExCommand* objects.

3.6 Macros

MacroManager was not needed as Macros are specific to this *VMModel* implementation, and so some of the methods I planned to implement as part of that class were ported over to *VMModel*. To implement nested Macros, a new class called *MacroInfo* was created that holds information about a specific Macro that is being played. *VMModel* owns a list of *MacroInfos* that can potentially be empty, and it grows as we keep nesting inside Macros.

3.7 Enumerations

The initial plan did not have anything mentioned about *Enums* as they were not really deemed necessary but while designing VM, I made a couple of *Enum* types to ensure that I always get data from within an expected set of values. Some of them have been mentioned in the UML while some non-trivial ones also exist in the code, for example: *WindowState* has an EnumType of Position, and Offset that helps it in modifying cursor positions, and calculating offsets.

4 Design

4.1 MVC architecture pattern

The MVC architecture pattern, though a little hard to understand at the beginning, was really useful in making the development process friendly and easy. It allowed parallel development, wherein I could easily switch between working on the logic, input and output. It also made it easy for me to update things later on. For instance, for the longest time I thought that I had perfected my printing logic in my *View* class only to find out just one day before submission that it was messing up output in case I had lines in file spanning across multiple lines on the screen. Fortunately, having the implementation of my View made it really easy for me to modify how the *WindowState* (File Buffer) was printed without letting it affect any other area. This is just one of the instances where MVC architecture allowed for easier addition of features and fixing of bugs.

4.2 Single-responsibility principle

All of my classes follow the single-responsibility principle, thus indicating *strong cohesion* and *weak coupling.* The purpose of each class is just one thing which can be summarised as follows:

- Model: Interaction between Controller, View and Modes
- View: Output information
- Controller: Input information
- Modes: Parse input
- Command: Modify state
- State: Hold information about file buffer
- History: Hold information about contents of state in different timeline
- FileManager: Read, Write, and Create files
- MacroInfo: Hold information about currently playing Macro

In my opinion, a good rule of thumb is that if the purpose of a class can be described in just a short sentence, it is a good indication of this principle being followed.

4.3 Open-closed principle and Strategy pattern

Features that might need changes or additions in the future are implemented with *abstraction*- some making use of purely abstract interfaces. This helps in making sure that my already implemented classes are *closed for modification* but still *open to extension.* For example, while I was developing VM, I was able to easily add the *ExMode* as I had modes implemented within *VMModel* as a collection of super-class *Mode* objects. This is an example of the strategy pattern, and this has been used in my structure of VM at multiple places like Models, Views, Controllers, Modes, Commands, and ExCommands.

4.4 Liskov-substitution principle and Template Method Pattern

Most of my abstract methods are wrapped inside non virtual public methods to follow the Liskov-substitution principle. This helps me maintain invariants for my classes and also control the entry and exit procedures for some methods.

The best example of this is the *execute* method in my *Command* class that it public scoped. It calls within itself a private virtual method called *execute-For*, but before it does that, it does something at its entry and calls methods after its exit. This proved essential in avoiding code duplication for commands as I could just make the call to *memoriseState* if the command being run was *undoable*, and manipulate the number of times that command was being executed based on whether or not it was *countable*. On exit, I was easily able to notify my *VMModel* to update view for the current active state.

4.5 Dependency Inversion Principle

None of the high-level classes depend on low-level module within my implementation of VM. Each class either depends on a virtual abstraction or low-level modules overload from high-level modules with strong sanity checks.

4.6 Observer Pattern

Observer pattern is a quite useful pattern when building programs that change output based on input and display information to the user in real-time. As expected in the MVC pattern, my view and model exist in a observer relationship. Model notifies the view with a state, that it needs to update itself. *VMModel* acts as the subject in this case as it adds views to the Model class so that it can easily be updated in the future when needed.

4.6.1 Pimpl Idiom

All of my abstraction works on the basis of pointers, and therefore, this assists me in following the Pimpl Idiom. Since I have pointers to implementations, for example: In *VMModel*, I have pointers to the State, and Modes; in *NormalMode* I have pointers to all of the individual command classes (as pointers to superclass Command); and in *ExMode* I have pointers to all of the individual ExCommand classes. This allows me to use forward declarations instead of including header files for the classes, helping me *reduce coupling*.

5 Design Walk-through

While the previous page talked about the design in terms of its technical details, this section talks about the design from a walk-through perspective as to what happens when a user starts interacting with the program. Let's follow the process by breaking it down for more clarity.

- The VMModel is constructed by providing it the argument details, on the basis of which it create(s) *WindowState(s)* and then *start()* is called.
- Before start runs, *TerminalView, TerminalController and Modes* are constructed that are specific to this model- view and controller provided with a shared pointer to ncurses wrapper class *Terminal*.
- In Start, the view is loaded for the first time with the file and some information in the Statusbar. *View* takes care of this after model tells it to run.

- Controller provides the input as a sequence of keystrokes, and model passes it to the current active mode (*NormalMode* by default) for parsing.
- Each Mode tries to parse the input sequence in its own way, and for every successful identification of a valid sequence, the *WindowState* is modified.
- Whenever the *WindowState* is modified by the execution of a Command, VMModel is notified and active view is updated.
- Some commands can switch between Modes and for them, they call the *changeMode* method on the VMModel, view is again notified to update.
- Each command has its own *executeFor* method that uniquely modifies the *WindowState* based on what it is supposed to do. Some commands make use of helper methods as well, but they are private scoped to ensure no-other class can access them.
- When a Macro starts recording, a flag is turned on in the VMModel. Whenever *getInput* is called in the future on the Model, it checks the flag and puts the character in a vector if a macro is being recorded.
- When a Macro stops recording, the recorded sequence along with the macroName is parsed inside *stopRecordingSequence* method of VMModel, and *FileManager* is used to store the information in a hidden file for future use.
- When a Macro is played, a flag is turned on and a *MacroInfo* object is created and stored inside a vector in VMModel. *getInput* starts behaving differently till a Macro is being played, and when the vector of MacroInfos finally gets over, it again starts behaving the normal way.
- The *parseInputSequence* method in Modes has the ability to check if a command is recordable, or based on any other condition specific to them, and erase some part of the recorded sequence with VMModel's *partlyEraseRecordingSequence* if it is not.
- WindowStates implementation is an interesting aspect. It holds information about the current position of the cursor in the File, based on which the View positions the cursor on Screen. State also holds a vector of past and future instances of its content as *History* objects.

6 Resilience to Change

As I have already discussed heavily in previous sections of this document, my implementation is heavily resilient to change as it follows the *SOLID* principles of OOP and has *low coupling* and *high cohesion.*

Any future commands can easily be added by extending on the already existing super-classes for Commands, and furthermore, behaviour specific to type of Command can be modified easily by adding a subclass to Command for a new type. For example, currently Insert, Macro, Editing, Motion and CutPaste are the type of Commands that exist. This makes it really easy to add commands that depend on other type of commands. Any mode can easily be added by inheriting from the Mode superclass, and adding an instance of it to the VM-Model class. We can also add support for two displays/views by adding another View object in Model's collection of views. Since abstraction is used wherever there is a possibility of extension or addition, it is really easy to expand my VM in the future if need be.

7 Answer to Questions

7.1 Multiple File Support

My VMModel maintains a vector of *WindowState* objects, where each WindowState is a file buffer. This makes it really easy to add multiple file support, or rather made it so easy to add multiple file support that I already implemented it as a bonus fun feature. The user needs to provide all the files to open when launching the program with the *./vm* command. They can then switch between files by using the *:n* and *:prev* commands to go back and forth between them. Users can also add new files from within a VM session by using the *:e* command, optionally followed by a file name to open a new file in VM.

7.2 Handling only readable files

My implementation of VM has a totally separate FileManager class that provides static methods. When attempting to write to a file, we can throw an error from within that method and then update the status bar accordingly from wherever the *writeToFile* method was called from. I can then use my *getInput* method to get an input from the user about what to do- probably a y/n for yes or no regarding creating a new file. If we want a new file name, we already have the facilities for getting input from the status bar. We can then just call *createFile* with a different file name and then call *writeToFile* with that file name to write in the newly created file.

8 Extra Credit Features

These can be turned on by running VM with a *-enhanced* flag. My VM has support for multiple files, so if you run with the -enhanced flag, you can use commands like *:e [filename], :n, :prev* to take advantage of the multi-file handling. There is also an option to record macros with longer names as only register names in Vim are pretty useless when it comes to remember what a macro did, since there is no connection between macro purpose and register. You can save macros with alphanumeric names like *3down*, that make the cursor go down 3 times, etc. Similarly, you can playback macros with longer names than one character. I have also dodged managing my own memory by making use of standard memory containers like unique pointers and shared pointers. Another little fun command that gets enabled in enhanced mode, is the *:auth* command that I was using to differentiate between my VM and the actual Vim. Handling multiple files was really challenging, but because of my implementation throughout, I was able to convert a one file VM to multiple files very easily.

9 What would I have done differently?

I feel like I did a fairly good job with my implementation of VM as towards the end, when I was rapidly adding more features, I realised that because of my architecture and implementation, I was able to do so fairly quickly.

However, if I had the chance to start over again, I would have definitely thought about the part of making parent sub classes and using abstraction earlier as this time, I faced problems during my development process wherein I had to completely pause what I was working on and then try to implement abstract super classes to solve my issue. I started with my Due Date 1's plan of having just a VMModel, TerminalView and TerminalController class but soon had to make their more generalized super-classes. Similarly with Command types like Motion commands, Input commands, etc. I had made a lot of them before I decided to make sub-classes for each command type and batch up the smaller individual commands in those command type sub-classes.

I also would have had overloaded my method for command execution, making it take model reference and an integer for number of times it needs to be executed, or making it take an a model reference and an input sequence. If I was able to do this, I could have branched off my ExMode commands from my Commands super-class and this is also a reasonable thing to do as some commands would have also worked well if they received the input sequence instead.

Another thing I would have likely done is that I would have started working on this project early. I had a really lengthy assignment in my CS241E course because of which I got delayed in starting this project and towards the end I was facing shortage of time. I had to skip out on a really cool feature that I wanted to implement in my VM- an interactive interface for git that could use system commands to enable VCS from within VM.