2018-12-03
Felipe Bemfica
Isaac Jensen-Large

VM: A Vi/Vim Clone, Written in C++

Introduction

One of the first assumptions our team made going into this project was that Vim is a simple pipeline of commands. The user provides input, which is transformed into commands, which are performed upon the document, which is rendered onto the screen. We divided our program into parts under this assumption, but we were wrong. All of these cleanly-divided parts end up depending on each other for various features, and they become more closely coupled than we had anticipated in our original design. A more mature system of objects and interfaces was required than we had thought at first. VM is a study in how assumptions about design go out the window when theory is put into practice.

Components Overview

Model

The Model is responsible for the document. It executes commands upon the document, and it owns the data structures that comprise the document. It is part of the MVC architecture which we used to design the program.

The document itself is called the Buffer, and exists as a doubly-linked list of linked lists of strings. This data structure was chosen to provide fast insertions for large file sizes and for long line lengths. Responsiveness is a top priority in a text editor, and choosing a powerful data structure is important.

View

The View is responsible for the user interface. It has access to the Model, and formats data from the Model and about the state of the program in general it in a way that tells the user everything they need to know.

The View is comprised of two components, each of which handles one part of the View's responsibility. The UI is in charge of actually writing to the display. It abstracts away the ncurses functions behind the interface and exposes a set of simple commands to display information and to accept input from the user.

The Formatter is in charge of reading data from the Model and arranging it into the correct shape; it creates soft returns when lines become too long, it finds the appropriate viewport for any given cursor position, and it displays all the information comprising the status bar. The Formatter is registered as an observer to the Model, and when the Model makes

changes to its Buffer, the Formatter is notified, and updates the screen. The Controller, in turn, draws the user's input through a very thin interface with the view.

In this way, the Model and the Controller are kept almost entirely unaware of the View, leading to a clean design with low coupling.

Controller

The Controller is responsible for handling user input, and processing it into commands for the Model. This involves holding on to all the program state that is not part of the document itself. The Model does not need to be aware of the cursor position, or whether or not the user is in insert mode. These are concerns related to creating commands, not with the document. These are the concerns of the Controller.

The Controller owns two components which, together, help it handle its responsibilities: the Command Generator and the Session Manager.

The Command Generator is concerned strictly with transforming raw user input into commands. Initially, it had a lot of information associated with it, such as the input mode and the contents of the search bar, which other components, like the view, needed to be aware of. We believed that the Command Generator should not be responsible for informing other parts of the program about the state of the user's session, so we created an additional component, the Session Manager.

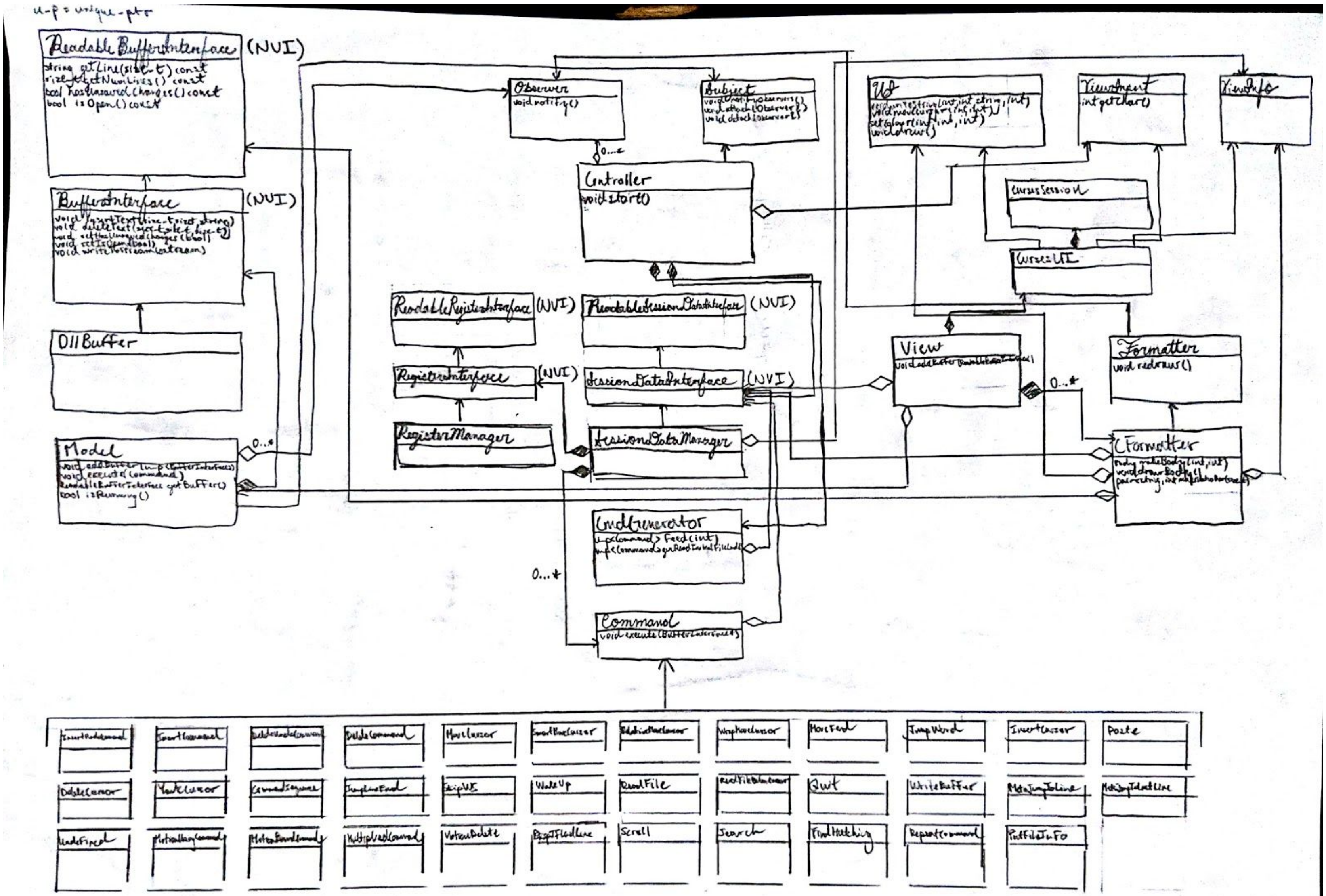
The Session Manager controls the cursor, the position of the viewport on the page, the registers, the search bar, and other VM data not integral to the document itself. It serves this information to the View, to individual commands, to the Command Generator, and to anything that needs access to it. It provides a clean and abstract way to share information that is required by other parts of the program without creating a mess of references to different components. By aggregating state and providing simple interfaces, the Session Manager cleans up a lot of the messiness that can come with this sort of information, leading to much looser coupling.

Lastly, the Controller itself runs the VM Session. This means that it is responsible for executing the driving loop of the program; it draws input and it forwards the resulting commands to the Model. The Session is the driving force behind execution.

Commands

The commands are one of the most complex parts of the program. The user is capable of interacting with the document in a variety of ways, and the commands need to encode all that functionality into one package. We decided that the best way to build flexible, extensible commands would be to build them using the Decorator design pattern. This enabled us to create multiplied commands, command sequences, and motion-enabled commands (such as 'dw') as simple Command objects. This way, commands are open to extension while remaining closed to modification.

Each command object is given interfaces to the Session Manager and to the Buffer; by leaving the connection between the Session and the Model in the hands of small, ephemeral commands, we keep the state of the Session and the state of the Model separate and very loosely coupled.


u-p = unique-ptr
ReadableBufferInterface (NVI)
string getLine(size_t) const
size_t getNumLines() const
bool hasUnsavedChanges() const
bool isOpen() const
BufferInterface (NVI)
DllBuffer
Model
bool isRunning()
Observer
void notify()
Subject
Controller
void start()
0...*
UI
ViewInfo
CursesSession
CursesUI
ReadableRegisterInterface (NVI)
RegisterInterface (NVI)
RegisterManager
ReadableSessionDataInterface (NVI)
SessionDataInterface (NVI)
SessionDataManager
View
Formatter
void redraw()
CFormatter
CmdGenerator
Command
Paste
Quit
Scroll
Search
ReadFile
WriteBuffer
WakeUp
FindMatching
RepeatCommand
PrintFileInfo
JumpWord
InsertCursor
MoveCursor
DeleteCommand

Differences Between the Proposal and the Final Program

The biggest and most obvious difference is apparent just from the names. Gone is the original Model-View-ViewModel architecture; we return to the classic Model-Controller-View. The rationale for this was simple: Although it initially seemed appealing to have the View just be a dumb window to display the information shown to it by the ViewModel, with all the logic being handled by the ViewModel, we came to believe that the responsibilities of formatting the contents of the Model for display and of turning user input into command were so distinct that the components responsible for them should not exist under the same umbrella. That represented low cohesion.

To this end, we turned what used to be the View into what is now the UI, and created a new View object that managed both the UI and the Formatter. We felt this separated the concerns of handling user input and displaying information far better than our old design. Furthermore, it meant a big increase in the cohesion of the Controller, and a big decrease in the coupling between the View and the Controller. The Controller is barely aware of the existence of the View at all any more, save for a very small interface through which it draws raw user input.

Another design-changing refactor involved gathering all the state related to the user's session in the Session Manager. It used to be that the Model was responsible for the cursor position, and the Formatter had to take the input mode directly from the Command Generator. The longer we worked with this design, the more we felt that the responsibility of handling this state was being mismanaged and spread out; by bringing all of it together under the Session Manager, we created a cleaner and more extensible design, with looser coupling and tighter cohesion.

And the Session Manager design is definitely more extensible. By the time we were developing the commands for moving around the display port (Ctrl-D, Ctrl-F, etc.), we had already made the switch to the Session Manager. Previously, the position of the viewport was the responsibility of the Formatter; for a command to modify that, it would need to be given access to the contents of the Formatter, which would poke a hole in our lovely abstraction and make our design much, much messier. But fortunately, the Session Manager had already solved this problem for us! By taking ownership of the position of the viewport, it obviated the need to give commands access to the Formatter, since commands already were reading information from the Session Manager. And since the Formatter already read information like the input mode from the Session Manager, it did not need any more privileges than it already had.

Accommodating Change

One of the biggest areas of extension in this program is the commands. They are the vector through which the user interacts with the program, and as such, almost any extension to the program is an extension to the commands.

With this in mind, we designed commands that are very open to extension while remaining closed to modification using the Decorator design pattern. Meta-commands are commands which are built using other commands as components. The subordinate commands are then intelligently executed by the Meta-command, leading to simple, logical constructions. For instance, a multiplied command, which executes its sub-command a given number of times in sequence, or a motion-bound command, which is given a motion and a command, and executes the command on the text that is traversed during the motion.

One of the questions in the initial proposal was interesting enough to us that we considered adding it as an extra feature. Although we did not have time to implement it, we worked from the beginning to make support for multiple files a possible extension. By making the Model distinct from the buffer, and by linking the formatter to the buffer, we created a design that was prepared to be extended: We planned to give the Model an array of buffers, and to have the Model execute commands upon an active buffer, which could be changed on demand. Each buffer would have its own formatter, and, since only the formatter attached to the active buffer would be notified, the screen would be continually updated to show the user only the active buffer.

However, this would also be achievable with only one formatter. The real use of multiple formatters is in anticipation of multiple viewports into multiple documents. One of our favourite features of Vim is side-by-side viewing of multiple documents; by having multiple formatters writing onto different parts of the screen, we could achieve this with minimal changes to our design. A big goal of our design was to be very extensible in the direction of multi-file support, and although that feature was never implemented, it remains a big component of the design philosophy, and of our commitment to flexible design in general.

What lessons did we learn about working in teams?

Our team reached the end of our project with newfound respect for Agile methodologies. Although the project specifications never changed, our own lack of experience with object-oriented design meant that, as the project continued, we found better, cleaner, more efficient, and more SOLID ways of doing things than we had originally planned for, and considerations we had not anticipated stood as counterexamples to many of our initial assumptions. If we had been tied to our initial design document, our final product would have been much worse.

And in order to revise a design, it is important to have very strong communication. Because of differences in our schedules, our team was sometimes able to meet up to discuss changes to the design, and sometimes not. When we did not, our understanding of the project suffered, and work slowed down as we came to grips with the ways in which the structure had changed. In order for a team to stay on its feet amid changing requirements and constraints, it is important to regularly meet up. This is the strength of team stand-ups: a quick meeting to make sure everyone is on the same page can make a big difference to teamwork.

On the other hand, we learned a lot about the challenges of designing something as a team. Object-oriented design is, in many ways, a subjective discipline, and it can be difficult to

settle disagreements when the criteria for which design is better are subjective, and can often only be seen in hindsight. When disagreements do crop up, it is important not to get stuck on them. Although sometimes there is a better choice and a worse choice, often the practical difference between design paths is small, and it is important to manage conflict during design sessions and avoid unnecessary arguments.

What would we have done differently?

As can be seen in our section about the differences between our original design and our final design, we learned a fair amount about object-oriented design over the course of this project. If we were to redesign the project, it would look much more like our final product, rather than our initial draft.

However, our biggest regret about the way we developed this program stems from the design of the Buffer. We implemented the Buffer as a doubly-linked list of linked lists of strings. This data structure is capable of inserting lines and text very quickly into long documents or long lines. In the long run, some kind of high-performance data structure would have been necessary for a text editor like ours.

However, it was not necessary for us. Developing the Buffer in this way was much more complex than a simple vector of strings, and it took much more time to make. This took time away from more sensitive parts of the project that we would have liked longer to work on. We prematurely optimised, and as a result, we did not have as much time as we would have liked to dedicate to implementing the functionality that was asked of us in the first place. There are more corner cases that we could have straightened out, and more testing that we would have liked to do. However, prematurely optimizing our data structures represented a mismanagement of time on our part, and is probably the biggest mistake we made in the development of this application.

Conclusion

Both members of our team are big Vim users, and although we have used the application extensively, we never understood what was happening behind the scenes. This assignment offered us the opportunity to pull back the veil, so to speak, and get our hands dirty with the design behind one of the applications we use most. It was a fantastic opportunity to better understand good, SOLID design, and to understand how to work as a team. We enjoyed the project a lot, and we’re glad that we had the opportunity to create a real application, instead of just completing an exercise.

Appendix

How would you support multiple files at once?

How this question affected our final design is detailed above; our original answer to this question follows below.

To support multiple Buffers, the Model would hold references to more than one Buffer. The CmdGenerator would be aware of which Buffer was currently being edited, and would supply each Transaction in turn with an interface to the appropriate Buffer. Each Buffer, upon being modified, would notify its respective Formatter, which would draw the new version of the file to the
screen. If the active Buffer were changed, the CmdGenerator would start feeding a different Buffer interface to its transactions, modifying a different Buffer and causing a different Formatter to draw different data to the screen. Simply displaying a different file would be done by calling a "wake up" method on the Buffer, which notifies the observing Formatter but does not exact changes on the Buffer.

How could you notify the user that they are editing a read-only file?

Similarly, we have included our original answer to this question too, and although our design was not influenced as much by trying to implement this functionality, the answer still stands.

Suppose we wanted to warn the user, but allow the user to edit the document while preventing them from saving. Information about the permissions of the file in question would be stored in the SessionManager, as it already has access to the file name. Then, transactions which mutate the file would inherit from a parent which warns the user upon trying to edit a read-only file, and then flips a bit in the SessionManager so as not to warn the user a second time.

Upon trying to save, the special transaction which would perform this would notice the read-only status of the file and would refuse to save, throwing another alert at the user. Conversely, if we wanted to prevent users from editing a read-only file at all, we would similarly make Transactions which mutate the Buffer inherit from a parent Transaction which would refuse to act on a file with a read-only bit.