VM Final Design

Overview

vm has the following directory structure:

./vir
- build
  - CMake stuff
  - vm executable
  - log.txt
- info
  - Logistics (uml, plan, etc.)
- src
  - include
    - headers
  - controller
    - controlller.cpp
  - input
    - command_parser.cpp
    - insert_parser.cpp
    - normal_parser.cpp
    - parser.cpp
    - replace_parser.cpp
    - rsearch_parser.cpp
    - search_parser.cpp
  - model
    - cursor.cpp
    - edit_tree.cpp
    - model.cpp
  - util
    - utilities.cpp
  - view
    - curses_window.cpp
    - split.cpp
    - tab_bar.cpp
    - tab.cpp
    - tool_bar.cpp
  - vm.cpp

The build and info directories are self-explanatory, but our src directory was structured to follow c++ project best practices (keeping headers in the include directory) as well as the MVC architecture. The controller hosts the run logic for our application, the model directory contains code that tracks the file, the history of the file, and meta-data about the cursor. Input contains an abstract parser class which is responsible for decoding and executing commands from the controller, util contains basic utility functions such as monus, log, etc. Finally, view contains ncurses wrapping as well as an object hierarchy representing the different types of windows we can display.

Design

We designed the project following the guidelines of the MVC architecture. Our model serves as an abstraction for the text file. The view uses the ncurses library to display a user interface in the terminal. The controller serves as an interface between the model and the view and notifies each of changes based on incoming requests (keyboard input). The controller parses input using a Parser class. The parser, as a component of the controller, notifies the model of any changes that need to be made based on input, and returns a value to the controller. Based on this value, the controller updates the mode its in and notifies the view of any changes that need to be made to the display.

Recognizing that only the model should have the ability to modify itself and its meta-data, we equipped the model class with a plethora of moveXXX commands that handle the many movement options that vim provides. For example, moveLeft, moveRight, moveRightI, toEOF, etc. The model also exposes methods that were heavily inspired by the iterator design pattern such as: setRange(Cursor c1, Cursor c2, const std::string &s), delRange(Cursor c1, Cursor c2), yankRange(Cursor c1, Cursor c2). The idea being that the Cursor serves as a position in the file, these functions then act on the range [c1, c2), and return a cursor to the appropriate location, as dictated by vim's behaviour. This allowed us to write essentially all of our normal mode commands as recursive compositions of model movement commands and model modifying commands.

To achieve low coupling, we made sure that the internal implementation of each component was independent of public use. We wanted each class to expose very general API that would allow for significant code reuse. To achieve high cohesion, we made sure that each class was more or less responsible for just one general thing. For example, the controller handles input and notifies the tabBar and toolBar of changes. The tabBar tracks each of the tabs and the Tab class handles the split(s) on each tab. The resizing of each component is done by the class that owns it. The splits handle the display of the data in the model and the model handles changes to the underlying file data. Each of the parsers are responsible for parsing in their individual mode and executing logic through calls to the model's API.

Resilience to Change

Our UI was designed so that they are easily modifiable and so that adding new UI elements would be rather simple. Everything UI related (status bar, tab bar, and main split in the current vm) inherits from an abstract base class called CursesWIndow, which is essentially a wrapper for window related functions in ncurses. The inherited classes override a *draw* method so that they can display what they need to. To create new UI elements, one would just have to create a class that inherits from CursesWindow.

Similarly, our parsers and modes were designed so that they can be modified individually and support the creation of additional modes. Each parser inherits a Parser abstract base class and overrides a function called *parse* to carry out their individual logic. They return State information so that the controller knows whether it needs to change the moder or to notify the UI elements of changes to the model. Thus, each parser can implement its own logic without having it affect any other parser. To add a new parser, one would just have to create a class that inherits from Parser. If any new functionality needs to be added, one would add to the State enumeration.

Although our syntax highlighter is currently hardcoded in the model due to time restrictions, it would ideally be a separate abstract class from which other classes can inherit. For example, a C++ syntax highlighter might be a child class that implements logic for highlighting c++ files. That way, new syntax highlighters could be added with ease and existing ones could be modified without breaking changes to other highlighters.

In addition, the design uses the MVC architecture to achieve low coupling. For example, the split (which draws the contents of a file to the screen) needs to know nothing about how the file is stored in the model. It gets the lines it needs through calls to *getLine(int line)* in the model's interface and renders these lines to the dimensions of the screen. Conversely, the model knows nothing about how the lines of a file are being displayed to the screen. Specifically, it has no knowledge of the screen width or height. It just stored the lines of a file as it reads them and returns them appropriately in its API. Other components, like the controllers or parsers, that need to modify the model do so through use of its public functions so that they also do not need to know the internal implementation of the model. The Cursor acts like an abstract iterator through the model. All the other components are similarly encapsulated so that a change to the internal implementation does not have breaking external ramifications, as long as the public functions still return sensible data.

Questions Part I

Question: *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?*

Vim allows this in 2 ways; either through tabs or through splits. Our goal was to support both of them, but we ran out of time to support splits.

To support tabs, we chose to have the view own a tab-bar of all current tabs that are open, along with an index indicating its current tab. This way, when we pass the view to the renderer, the renderer is able to draw the tab-bar if tabBar.size() > 1, and it will know the tab to draw in the middle part of the screen. Finally, commands such as :tabn, :tabe, etc. should change the displayed file, but again, this is not a problem in our implementation as splits own their files.

To support splits, we would need to change the renderer to support drawing of multiple windows, as well as have the controller track which split is currently in use. To support the drawing of multiple windows, we would need each ncurses window to store its coordinates and size, since ncurses cannot draw windows on top of one another. Finally, creating new splits would require the ability to resize all other splits affected by the splitting action, which would likely need a tree like structure so that splits know which other splits their sizing effects.

Question: *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?*

The first step in supporting this feature would be to check whether or not the file is actually read-only when it is opened. After some research, I found that you can pass a second parameter to fstream's constructor that allows you to specify the mode in which you open the file. Then, you could try to open the file in 'w+', and if that fails, try to open the file in 'r', updating the readOnly bool accordingly. If the file is readOnly, then it would be easy to emulate vim and warn the user that they are modifying a read-only file at their first modifying command.

To support the first option, we could disallow modifying commands by having a check in the implementation of ModifyAction.

To support the second, it would be easy to check if a split is executing a modifying action for the first time (by storing a bool on the split, for example), and mimic vim and warn the user when such a first action occurs.

Supporting both together could be done by modifying the warning message delivered by the second to become a yes/no question, asking the user if they wish to be able to make changes. If yes, then we proceed without the ability to save, if no, we block changes.

Finally, when the user tries to save, we will first check whether the file they are saving to has the proper permissions to save to the current file, and if not, we display a message (like vim) in the toolbar saying that the readonly flag is set, and we cannot save changes to this file.

Questions Part II

Question: *What lessons did this project teach you about developing software in teams?*

This project taught us that planning is crucial to the success of a team project. Had we not taken the time to properly design, plan, and delegate tasks for this project, we would have wasted effort on rewriting code that did not work together. Further, since more than one person worked on different components of the codebase that had to work in cohesion, we had to make sure that the functionality of public API was clear. In addition, the team aspect of this project emphasized the importance of having proper encapsulation so that code written by one partner can't be misused by the other. We found that if we took shortcuts by directly modifying classes, then we would introduce bugs into our code that took a long time to track down.

The project also stressed the importance of proper communication in a team environment. To use time efficiently, each partner should be well aware of what the other partner is working on so that no time is wasted writing duplicate code. For example, we kept a file called utilities.hpp which contained useful functions for general use (such as string manipulation) that reduce the amount of duplicate in our codebase.

Lastly, the project highlighted the importance of writing organised and readable code so that each partner could work on the other's code if required. To this end, we created a constants.hpp file to reduce the number of magic numbers and strings in our code. In addition, we made sure that functions and fields had clear naming and that there were no counterintuitive behaviors.

Question: *What would you have done differently if you had the chance to start over?*

If we had the chance to start over, I believe that we should have been more realistic with our design and plan. For example, we thought that we would have much more time to implement other features than we actually did. This was largely because we did not anticipate having to spend so much time on other courses during these past two weeks. Closer to the end of the project, we sacrificed proper design and organisation to ensure that we had a complete project. For example, if we had more time, the tab bar would ideally have been implemented using the Decorator design pattern since its display is conditional. In addition, the syntax highlighter, as mentioned above, would have been put into its own class. To make the model more cohesive,

we would have split it up into smaller helper classes that perform one function on the file (such as separate classes for search and movement).

Extra Credit Features

We used RAII to ensure that memory was properly managed. This was achieved through the creation of wrapper classes that handled the allocation and destruction of objects as well as the use of unique and shared pointers. In addition, we allowed for the creation of tabs, which allows for the editing of multiple files at the same time. In doing so, we had to make sure that the drawing of the view was completely self contained (so that external components would have no impact on what was being drawn) and only depending on the information in the model. That way, the creation of a new tab would just be creating a new split and associated model.