Final Report

Table of Contents

Introduction

Vim is bizarre. That is our overall conclusion after completing this project using C++. Although most features conform to some sort of fuzzy overall structure, we found that it is more often than not that exceptions to the rule forced us to slightly tweak our design to accommodate for the behavior of one single command and to allow the implementation to be extendible. Quantifiers are a classic example of this.

This report is a post implementation, high-level communication of our general design and implementation with a focus on changes in design in response to slight unforeseen technical challenges encountered during the implementation phase. Details covered include general architecture, resilience to change, answers to relevant questions, and extra features.

Overview

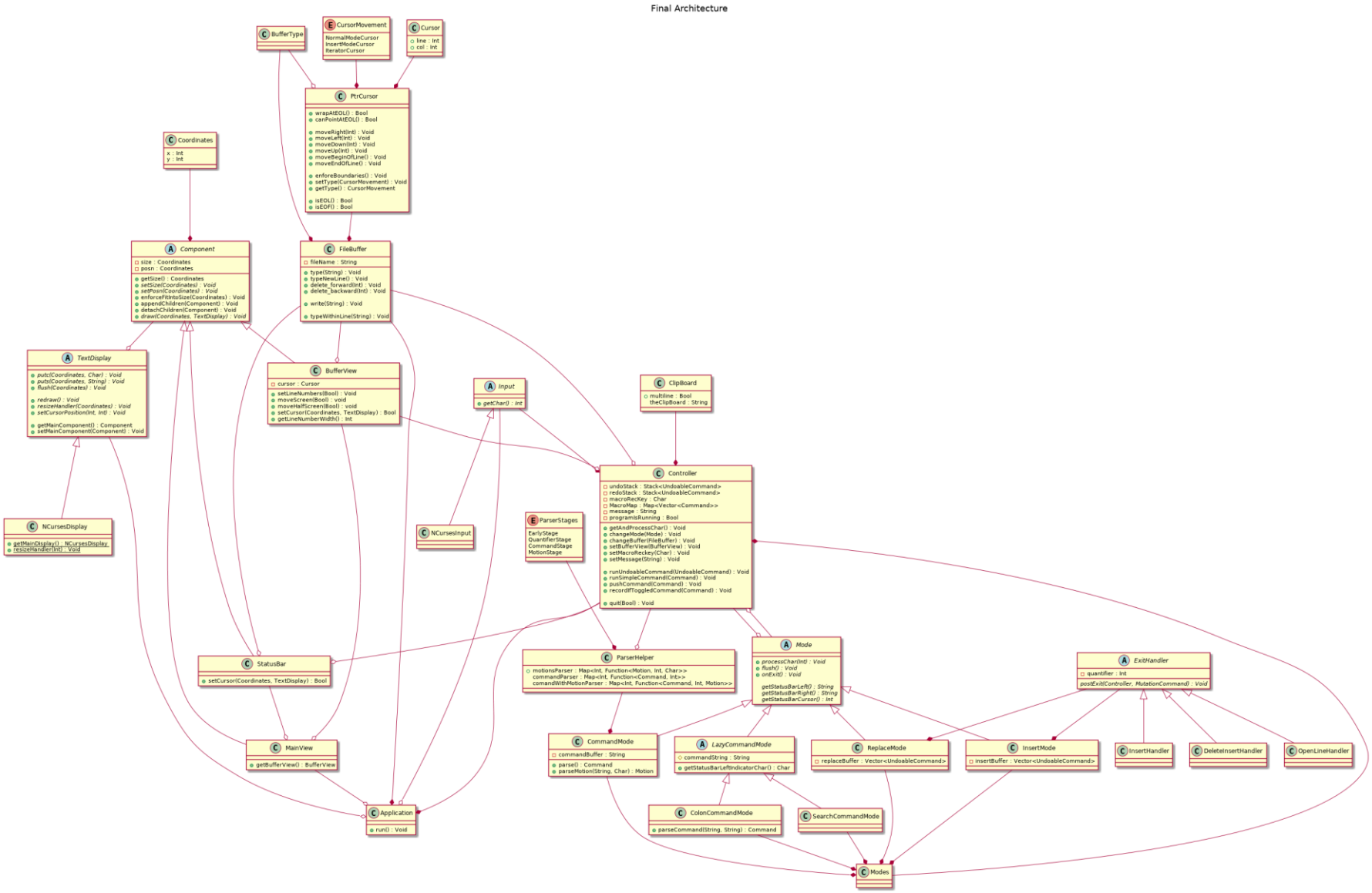
The main type of architecture chosen to implement our text editor is the MVC architecture. Due to the functionality heavy aspects of vim, the model is the most straight forward of the three to implement, with display coming in second and the controller being by far the most complex of the three.

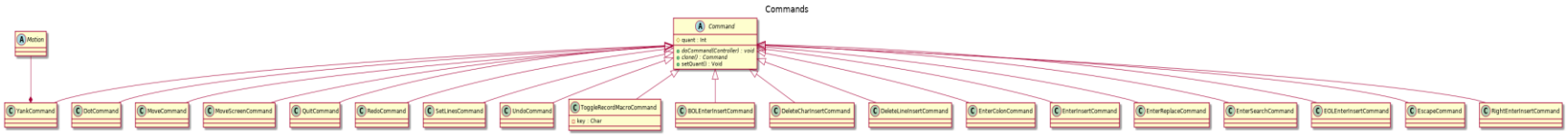
From the very beginning, the team identified the most difficult implementation challenges to be unbounded undos, commands which can be composed with arbitrary motions, macros, syntax highlighting, followed by quantifiers in general. Almost all of which are controller-based difficulties.

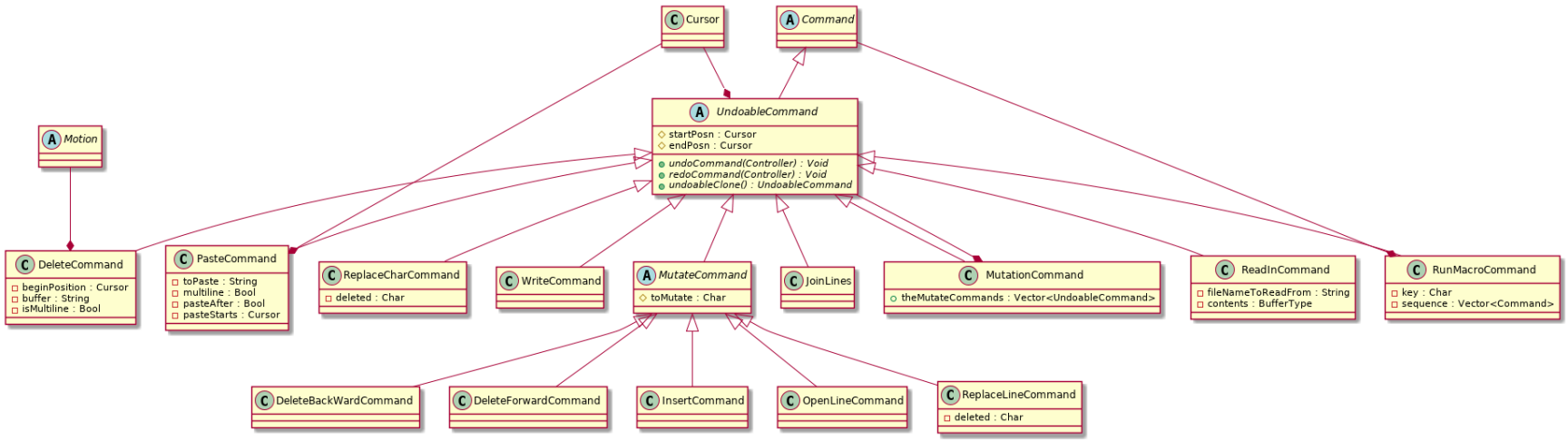
Keeping in mind that we may wish to extend the project beyond the scope of the requirements, the team spent most of the time allocated for design on the controller; keeping generality of our solution as a central goal and keeping resilience to change in the background of thought.

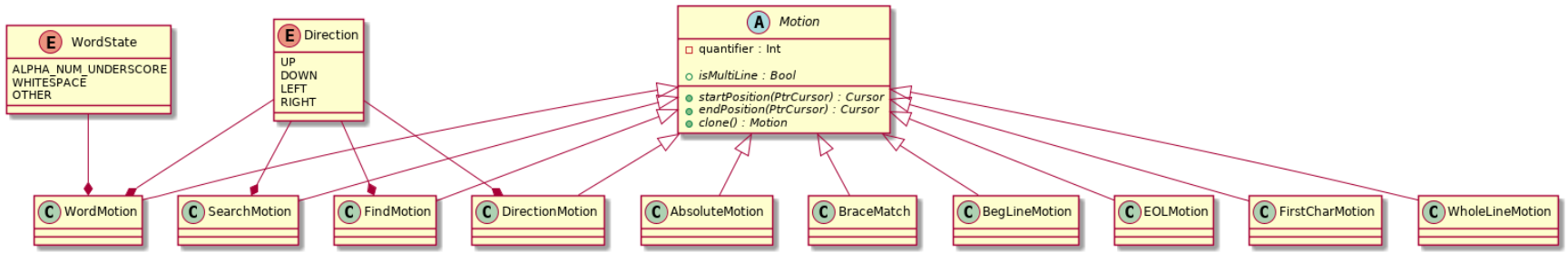
Due to the above, this report will explain out implementation of Vim with a major focus on the controller component, which is where most of the ingenuity lies.

Updated UML


Final Architecture
BufferType
CursorMovement
NormalModeCursor
InsertModeCursor
IteratorCursor
Cursor
line : Int
col : Int
PtrCursor
wrapAtEOL() : Bool
canPointAtEOL() : Bool
moveRight(Int) : Void
moveLeft(Int) : Void
moveDown(Int) : Void
moveUp(Int) : Void
moveBeginOfLine() : Void
moveEndOfLine() : Void
enforceBoundaries() : Void
setType(CursorMovement) : Void
getType() : CursorMovement
isEOL() : Bool
isEOF() : Bool
Coordinates
x : Int
y : Int
Component
size : Coordinates
posn : Coordinates
getSize() : Coordinates
setSize(Coordinates) : Void
setPosn(Coordinates) : Void
enforceFitIntoSize(Coordinates) : Void
appendChildren(Component) : Void
detachChildren(Component) : Void
draw(Coordinates, TextDisplay) : Void
FileBuffer
fileName : String
type(String) : Void
typeNewLine() : Void
delete_forward(Int) : Void
delete_backward(Int) : Void
write(String) : Void
typeWithinLine(String) : Void
TextDisplay
putc(Coordinates, Char) : Void
puts(Coordinates, String) : Void
flush(Coordinates) : Void
redraw() : Void
resizeHandler(Coordinates) : Void
setCursorPosition(Int, Int) : Void
getMainComponent() : Component
setMainComponent(Component) : Void
BufferView
cursor : Cursor
setLineNumbers(Bool) : Void
moveScreen(Bool) : void
moveHalfScreen(Bool) : void
setCursor(Coordinates, TextDisplay) : Bool
getLineNumberWidth() : Int
Input
getChar() : Int
ClipBoard
multiline : Bool
theClipBoard : String
Controller
undoStack : Stack<UndoableCommand>
redoStack : Stack<UndoableCommand>
macroRecKey : Char
MacroMap : Map<Vector<Command>>
message : String
programIsRunning : Bool
getAndProcessChar() : Void
changeMode(Mode) : Void
changeBuffer(FileBuffer) : Void
setBufferView(BufferView) : Void
setMacroReckey(Char) : Void
setMessage(String) : Void
runUndoableCommand(UndoableCommand) : Void
runSimpleCommand(Command) : Void
pushCommand(Command) : Void
recordIfToggledCommand(Command) : Void
quit(Bool) : Void
NCursesDisplay
getMainDisplay() : NCursesDisplay
resizeHandler(Int) : Void
NCursesInput
ParserStages
EarlyStage
QuantifierStage
CommandStage
MotionStage
StatusBar
setCursor(Coordinates, TextDisplay) : Bool
ParserHelper
motionsParser : Map<Int, Function<Motion, Int, Char>>
commandParser : Map<Int, Function<Command, Int>>
comandWithMotionParser : Map<Int, Function<Command, Int, Motion>>
Mode
processChar(Int) : Void
flush() : Void
onExit() : Void
getStatusBarLeft() : String
getStatusBarRight() : String
getStatusBarCursor() : Int
ExitHandler
quantifier : Int
postExit(Controller, MutationCommand) : Void
MainView
getBufferView() : BufferView
CommandMode
commandBuffer : String
parse() : Command
parseMotion(String, Char) : Motion
LazyCommandMode
commandString : String
getStatusBarLeftIndicatorChar() : Char
ReplaceMode
replaceBuffer : Vector<UndoableCommand>
InsertMode
insertBuffer : Vector<UndoableCommand>
InsertHandler
DeleteInsertHandler
OpenLineHandler
Application
run() : Void
ColonCommandMode
parseCommand(String, String) : Command
SearchCommandMode
Modes


Commands
Command
Motion
YankCommand
DotCommand
MoveCommand
MoveScreenCommand
QuitCommand
RedoCommand
SetLinesCommand
UndoCommand
ToggleRecordMacroCommand
EOLEnterInsertCommand
DeleteCharInsertCommand
DeleteLineInsertCommand
EnterColonCommand
EnterInsertCommand
EnterReplaceCommand
EnterSearchCommand
EOLEnterInsertCommand
EscapeCommand
RightEnterInsertCommand


Cursor
Command
UndoableCommand
startPosn : Cursor
endPosn : Cursor
undoCommand(Controller) : Void
redoCommand(Controller) : Void
undoableClone() : UndoableCommand
Motion
DeleteCommand
beginPosition : Cursor
buffer : String
isMultiline : Bool
PasteCommand
toPaste : String
multiline : Bool
pasteAfter : Bool
pasteStarts : Cursor
ReplaceCharCommand
deleted : Char
WriteCommand
MutateCommand
toMutate : Char
JoinLines
MutationCommand
theMutateCommands : Vector<UndoableCommand>
ReadInCommand
fileNameToReadFrom : String
contents : BufferType
RunMacroCommand
key : Char
sequence : Vector<Command>
DeleteBackWardCommand
DeleteForwardCommand
InsertCommand
OpenLineCommand
ReplaceLineCommand
deleted : Char


WordState
ALPHA_NUM_UNDERSCORE
WHITESPACE
OTHER
Direction
UP
DOWN
LEFT
RIGHT
Motion
quantifier : Int
isMultiLine : Bool
startPosition(PtrCursor) : Cursor
endPosition(PtrCursor) : Cursor
clone() : Motion
WordMotion
SearchMotion
FindMotion
DirectionMotion
AbsoluteMotion
BraceMatch
BegLineMotion
EOLMotion
FirstCharMotion
WholeLineMotion

Design

The Design section of the report will communicate abstractions built around each individual component of the MVC architecture and have and additional section which addresses the cross-component issues encountered and solved.

Model

The model simply serves as a buffer between the user's edits and the actual process of writing to files. The data structure would load the contents of each file into memory upon opening a file and be continually updated through user edits. It would be the least flexible aspect of the project in terms of adapting to new input or output as it is purely an internal state tracker. Our original design used a double linked list of doubly linked list of characters as that would be the most efficient in terms of insertion and deletion. However, upon our initial building of abstractions around common operations, we decided to forego the premature optimization and employ a vector of strings to better leverage the standard library features for strings.

One of the most unexpected challenges lied in the cursor position. Vim allows the cursor to extend beyond the end of the line in certain modes such as insert mode while forbidding it from crossing that border in normal mode. Furthermore, most operations on the model would be relative to the current position of the cursor. To address this crucial component, we represent the cursor through a PtrCursor object which the file buffer "owns". It would have common operations such as moving left one character, up one line, or to the end of the line, most of which would be slightly different depending on the current "state" of the cursor, represented by an embedded enum class.

Additional abstracts such as write out or read in operations are also possible through the model but details are omitted due to their relative simplicity.

View

The View loosely resembles an abridged version of the Observer Pattern coupled with the Decorator Pattern. Changes to the file buffer is reflected in an appropriate manner for each stroke of the keyboard. However, we were inspired to generalize our design in a way similar to the popular ReactJS library for web applications. Our priority was to ease the transition into writing a true graphical user interface if necessary. Although the design is more refined than our initial planning, the overall structure did not change too much.

The most basic building block in the View is an abstract Component base class. Which is declares some shared aspects of every view object such as retrieving the size of the component, child components, getting the size of the parent component, attaching children, and whatnot. We then compose different components with differing responsibilities such as the status bar which reflects the state of the controller among other things and an abstract TextDisplay, which reflects changes in the model. The concrete implementation involves a wrapper class for the ncurses

library which “is-a” TextDisplay, but clearly this can be generalized to other libraries. Note that this elegantly solves any resizing issues as the root parent class would restrict the size of each of its child components, which would relay the information for resizing downwards, etc.

Although the base parent Component is “owned” by a wrapper Application object, we built an additional layer of abstraction acting as an interface between the model and view, plus the controller and view. This additional feature which we did not foresee facilitates the scrolling commands such as scrolling one screen down or half a screen up.

A note about syntax highlighting will be included further into the report.

Controller

By far, vim’s rich set of commands and ability to aggregate different commands is its biggest selling point. However, it is also the biggest headache for the implementor. Consider the following example: “3c3fA”. This sequence of characters, when pressed in the default command mode, would cut 3 times, from the current cursor to the 3rd occurrence of the character ‘A’ on the line. However, somewhat counterintuitively, the entire length of the cut over the 3 iterations would be placed onto the buffer and not just the last cut command. Even more of an issue is the undo command. As almost every command which mutates the state of the file buffer is undoable, it suggests the necessity of a general solution to the desired functionality of commands. Again, out final design is much more refined than our fuzzy initial planning, but no major changes had to be made.

Commands

As aforementioned most if the designing of Vim is centered around commands. We employ the Command Pattern to encode each and every valid command into a subclass of the abstract Command class. An abstract subclass called UndoableCommand represents any mutation done to the model. To address the issue with undos, UndoableCommands are pushed onto a stack whenever executed, and popped from when an undo is issued, permitting unbounded undos. The UndoableCommand class employ Non-Virtual Interface to add some pre and post command hooks such that storing the current position of the cursor before and after executing a command. Upon calling the undo or redo, we would navigate back to the final and initial position respectively on the file buffer before the logic is run.

Motions

A very cool feature of vim is that commands can be couple with motions. To resolve this, we introduce an arbitrary Motion class. Each subclass implements a nextPosition function which accepts a Cursor object as input and output where the nextPosition should be. If composed with a command, we would simply need to do the appropriate operations in the range between the current cursor position and the ending position, making for a very versatile design.

Macros

Of the two additional features to implement, the team decided to pursue Vim macros. To facilitate “recording” commands, the doCommand function for each command should never be called directly, instead, the controller has functions such as runCommand, etc which accepts a Command object and if the macroRecKey character is non-zero in the Controller, will make a copy of the command, push it onto a vector of commands in an appropriate map of such vectors, then actually execute the command. To run a macro, we would simply retrieve the appropriate recorded vector of Commands from our “Command Cache” and push a wrapper class onto the UndoStack.

Note that this requires us the implement the Virtual Clone Pattern for Commands as well as Motions as we have no knowledge of the subclass type. The team employs the Curiously Recurring Template Pattern and created a very cool template class called Clonable in the utility folder which accepts three types as argument and generates the appropriate base class for any concrete class, automatically implementing the clone method.

Mode

Although our Command structure is well though-out, we do not have a way of interpreting each key press. For this task, we introduce the abstract class Mode, which declares a getChar function which generates different commands and executes them depending on the type of Mode and state of the previous key presses. On a side note, our implementation of the parser for command mode is very cool. It involves a map of lambdas and you are encouraged to take a peek.

One unforeseen issue is that there are multiple ways of entering InsertMode, which results in the different additional side effects depending on the key pressed and the preceding quantifier. For example, the ‘O’ key would insert a newline and another copy of the characters typed during insert mode *n* number of times, where *n* is the multiplier preceding ‘O’. To address the, we implement an additional abstract ExitHandler class. The appropriate command to enter the InsertMode would also set the mode’s ExitHandler, which will be called upon exiting the mode.

Input

Perhaps one of the more mundane cogs in the Controller machine is input. We abide by the Liskov Substitution Principle and have the Controller own an abstract Input class. For this specific instance, we would simply use the NCurses input function but clearly, this can be generalized to use other libraries.

Application

The application simply serves as the owner of the controller, model, and view components with the addition of interfacing with the command line arguments.

Resilience to Change

Separate Compilation

As our implementation grew larger and larger to over 120 files, we found that compilation times grew exponentially (not literally). We were able to reduce this greatly by forward declaring classes and declaring pointers, unique pointers, and raw pointers in the header files. A prime example is the Controller class header file. There are 12 forward declarations in the header.

On hindsight, this greatly improved our production speed, as we did not have to wait the full 30 seconds each time fixing a small bug or making an incremental change.

Coupling

Coupling is a measure of how strongly different modules depend on each other. We strive to decrease this as much as possible. Both our Model and View modules implement abstract base classes which only determine an interface and not any hint of implementation. Assuming two completely different, but correct and conforming implementations of either the Model or View hierarchy, there should be no issues interchanging the two.

Cohesion

Cohesian is a measure of how closely elements of a module are related to each other. Ideally, each element should perform one task. This is extremely well demonstrated in both the Controller and View modules.

Each Component in the View can be arbitrarily composed on top of each other, so it is paramount that each Component only reflects one type of state. For example, as previously mentioned, the StatusBar component reflects messages to be display from the Controller and that is all.

On the other hand, the Controller is divided into Input, Mode, and Commands, which are each responsible, on a high level, for retrieving, interpreting, and carrying out commands. It is easy to see the atomicity of the design.

Beyond the Scope of the Requirements

One of the team's priorities in designing and implementing this project is the ease with which we can extend the project beyond the course. Questions such as: "Is it possible to add even more commands?", "Can we use a graphical user interface instead?", and even "Would optimizing the function logic be easy?", were foremost on our minds.

To address these valid concerns, we determined what is means on an abstract level to be a Command, a Component, and tailored our implementations upon interfaces rather than specific

implementation. This resulted in an extremely versatile and adaptable project, easily extendible beyond the scope of the requirements.

Answers to Questions

Multiple Files

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?

Answer

Our design allows for the Controller and View to switch seamlessly between file buffers and still retain things such as macros. The only issue is that undos would not be stored in the controller but on the file buffer for each file, which is slightly unintuitive. Another solution would simply be to create a separate Controller, View, Model for each file and swap between them from the Application level, however, we would then lose our ability to transfer macros between files, which is undesirable.

Write Permissions

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?

Answer

Upon opening a read-only file, there should be no issues with mutating our representation of the file buffer itself, However, when we attempt to write to the file, should the output stream file fail for any reason, it would reasonable to throw an exception which is caught ideally in the Mode which interpreted the keystroke. This would implement the second of the two propositions.

To account for the first, we would try to open the file for writing upon loading it into memory for our model. If it fails, any attempt to issue an UndoableCommand should throw and exception.

Extra Credit Features

Macros

Between the two additional implementation challenges, we chose to implement macros. The actual running and recording macros logic did not take long to implement, however, the entire infrastructure to support this ability were of absolute importance. Please see the Controller section of the report for further discussion of macros.

Syntax Highlighting

Due to time limitations, we were unable to implement syntax highlighting. Despite this, our design could easily accommodate for an additional Component in the View which processed each line for possible matches to C++ keywords and manipulated the view library (in this case ncurses), to print colored version of this words instead.

Redos

In real Vim, we are able to "redo" commands that were previously undone. We decided to implement this feature by having an additional RedoCommand function in the interface for UndoableCommands. The issue with this feature is potentially the amount of code duplication in the DoCommand and RedoCommand function implementations. To accommodate for this, we separated the DoCommand function into side effects such as remembering which character we just deleted and pure functional aspects which are shared between the redo and do commands. The doCommand by convention applied side effects, then calls a "pure functional" helper while the RedoCommand only calls the helper. This convention greatly reduced the amount of code duplication necessary and made debugging much more simplistic.

The actual inner working of redo are simply that the undoStack is moved to the top of the redoStack upon issuing an undoCommand instead of being discarded and we then we would call the redoCommand function of the top most object in the redoStack, then moving it right back to the top of the undoStack when a RedoCommand is issued.

It is interesting to note that the undo and redo commands are actually **not** undoable as they only manipulate the undo and redo stacks.

RAII

There are no instances of the keyword "new" or "delete" applied for the purposes of memory management in our entire application.

Final Questions

Team Software Development

Question

What lessons did this project teach you about developing software in teams? If you worked

alone, what lessons did you learn about writing large programs?

Answer

Before commencing the project, it was sort of overwhelming to consider the magnitude of the assignment laid out before us. It was normal that we rushed ahead tackling imaginary barriers in our minds before laying out a complete design. However, having some experience writing software in teams, we spent extra time in the design phase to make sure both team members were satisfied with the overall design. In the addition, this helped create a natural delegation of implementation, as members who had a clearer idea of what the view should look like chose to work on the view while, members with more opinion on the implementation of commands were responsible parts of the controller.

One of the most important aspects of the project was communication. Both members of the team communicated with each other before and after making even medium scale changes to the code base. Due to this, we were more or less able to avoid merge conflicts in the repository and also gained extra insight into how to make our implementation more robust and legible.

A small but important trick was leaving TODO and FIXME notes in places where we decide to leave a temporary implementation or simply blank. This allowed us to make small incremental commits, which were easily reversible if necessary and also helped team members find where to search for bugs with a simple git grep command.

Hindsight

Question

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

Answer

Section which repeatedly required refactoring were the Command hierarchy and the PtrCursor object.

It was difficult to come up with the appropriate interface for RedoableCommands as narrowing down what logic was required for every command was difficult. With the hindsight available now, I would start with the interface of the RedoableCommand Hierarchy and reduce a great chunk of (necessary) time lost.

The cursor for Vim is always a headache, as the rule for being on the end of the line seems very arbitrary for some. Our solution kept a “state” of the PtrCursor as a value of a enum class forced us to continually set the type for commands, since operations on the model usually expects us to be able to traverse one of the end of the line, but most of the operations are executed in command mode, where the cursor is normally **not** allowed to do so. There is most likely a better solution than our current one, but it evades our minds even to this day.

Conclusion

All in all, it was extremely satisfying to employ the concepts which were taught in class in a medium size project. Although co-op allowed the team members to gain experience contributing to a codebase, it is completely different taking complete co-ownership over a non-trivial project.

The team will definitely be looking forward to using the techniques learned in this course in other courses as well as the industry.