1 Overview

The overall structure consists of a src folder containing main "Closures", which together parse the command arguments (such as flags and files), setup ncurses, initialize the models, views, and controllers from lib, and create the connections between them. These Closures are collected in a `Main` struct, which has a main() method to start the main loop. In the actual main function, I initialize a Main struct, check if we need to print a help message, and if not, then execute `main()` in a try catch clause, returning 1 if an exception is caught. Finally, for cleanup all Closures are RAIId so that destructors will cleanup the objects and processes they initialized (for example, `src/init_ncurses.h` runs `endwin()`).

The main loop starts by rendering the view (window, cursor, statusbar), waits for the user to type a key, then notifies the keyboard's observers. This repeats until the root window is closed (e. g. by `:wq`). The user's input is converted by the `UWSEKeyboard` class (i.e. the view in MVC) into a `KeyStroke`, which is then consumed by a `ModeManager` (i.e. the first part of the model in MVC). This manager contains the current `ModeType`, and forwards the keystroke to the `CommandParsers` corresponding to the current mode. The parsers will turn keystrokes into `Commands`. For example, `ComboNMParser` turns `4d2w` into `ComboNM{{4,'d','\0'},{2,'w','\0'}}`. Once a parser fully parses a `Command`, they notify their observers (i.e. `CommandRunners`) which update the rest of the model as needed. All of the things that runners will modify are:

- `ModeManager` - contains current `ModeType`. Also will forward `KeyStrokes` from `MacroRunner`
- `LinedFilebuf` - opens contents of a file from its filename into an in-memory buffer with line-based editing operations. `FstreamLFB` is a concrete implementation using `std::fstream`
- `FileManager` - maps each filename to its corresponding `LinedFilebuf`
- `Cursor` - current location (line, col) of the cursor in the `LinedFilebuf`
- `Tab` - combines a `Cursor` and a `LinedFilebuf`. Provides operations to synchronize `Cursor` with the window pane, e.g. scrolling the window when cursor moves, and vice-versa
- `TabManager` - provides operations to move forwards and backwards in multi-file editing
- `Clipboard`, `MacrosRegister` - stores clipboard data and macro recordings, respectively
- `RootStatus` - stores current message and/or error-code. e.g. E37: file not written
- `Window` - contains a `TabManager`. Provides operations for split-screening
- `HistoryTree` - stores snapshots of a `LinedFilebuf`'s contents; provides `undo`, `redo`
- `HistoryManager` - Maps filenames to their corresponding `HistoryTrees`

The view consists of a `NCWindow` which overrides the `render()` of `Window` to render in `ncurses`. It has 2 helpers: `StatusBar` and `Textbox` which render the current `Tab`'s cursor location, and the contents of the `LinedFilebuf`. The view also has a `NCCursor` which renders the on-screen cursor. Its position is calculated based on the `Cursor` and `Tab` by `Main` via the `CursorClosure`. Finally, the view has a `RootStatusRender` which renders a `string` at the bottom left of the screen. The message it displays is calculated from `RootStatus` by `Main` via `StatusBarClosure`'s `render()` method.

2 Updated UML

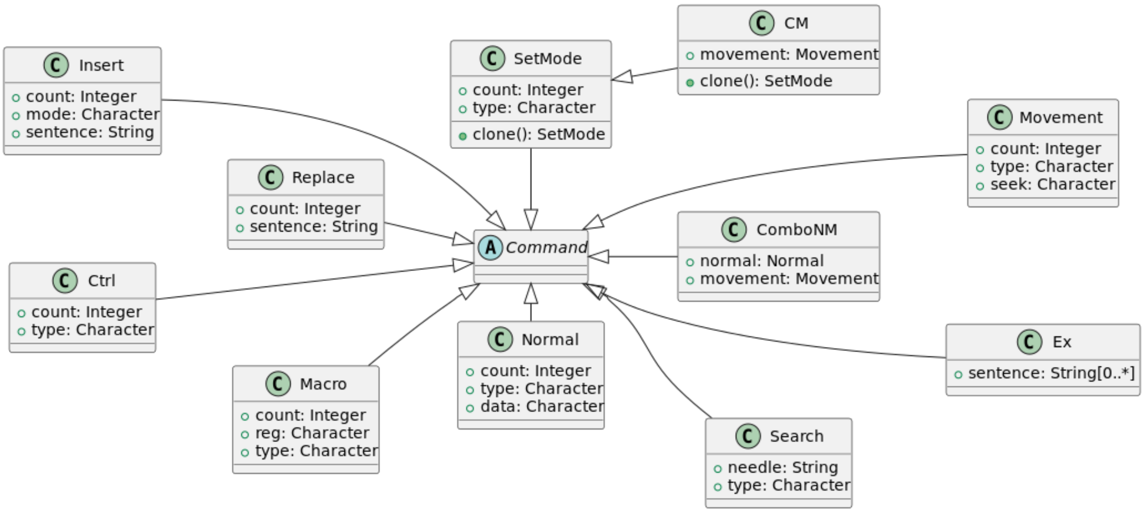


Figure 1. Command Heirarchy updated to include `CM`, `Search`, and any renamed fields.

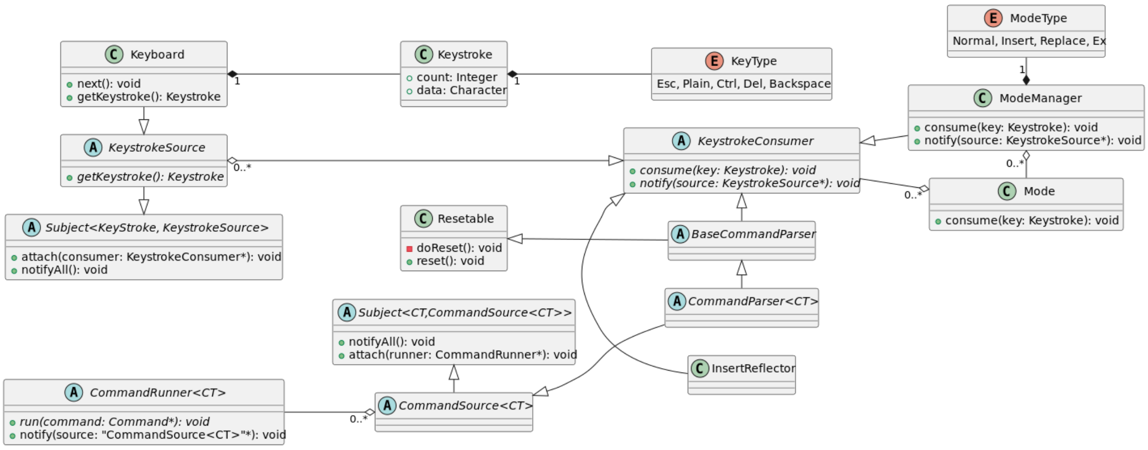
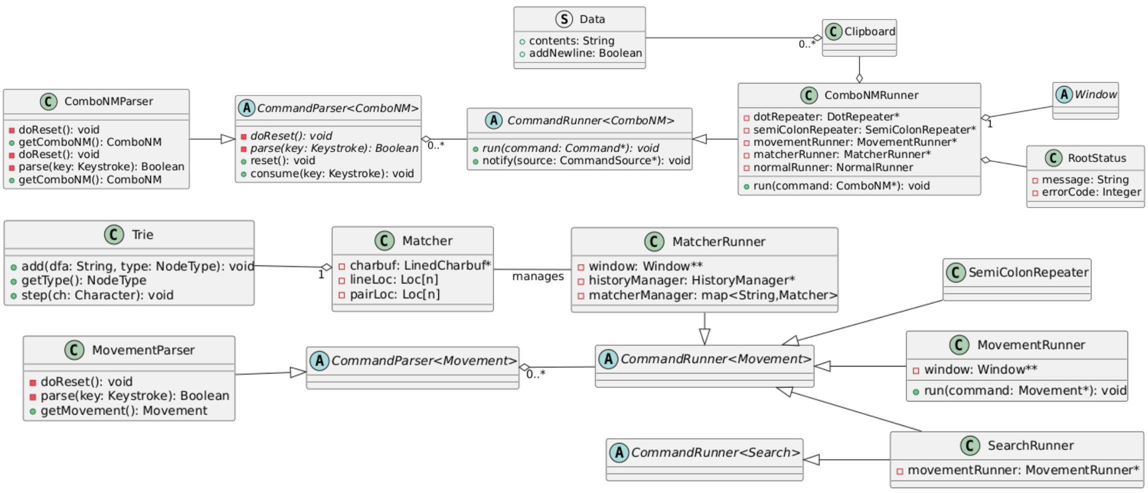


Figure 2. Subjects and objects for `KeyStrokes` and `Command`, including their templates (see **Observer Patter**)



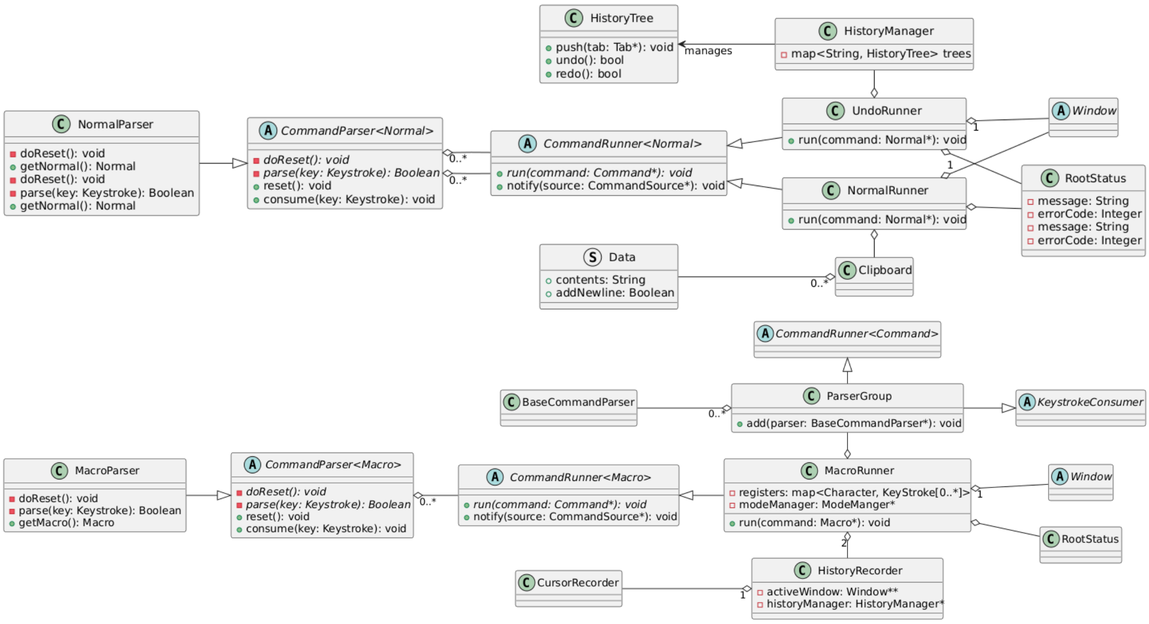


Figure 3. Some selected parsers, runners, and model classes for implementing `ComboNM`, `Movement`, `Normal`, and `Macro` commands, in order from top to bottom.

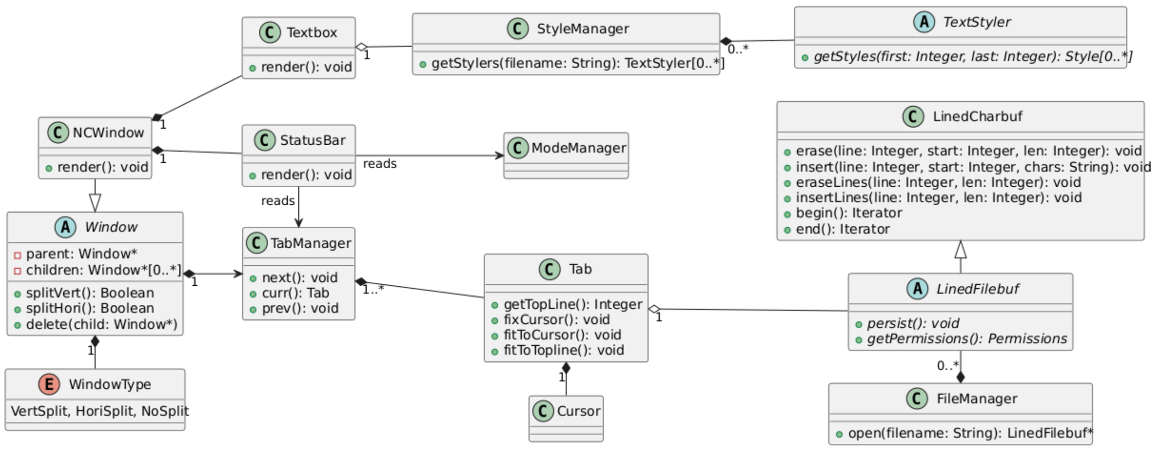
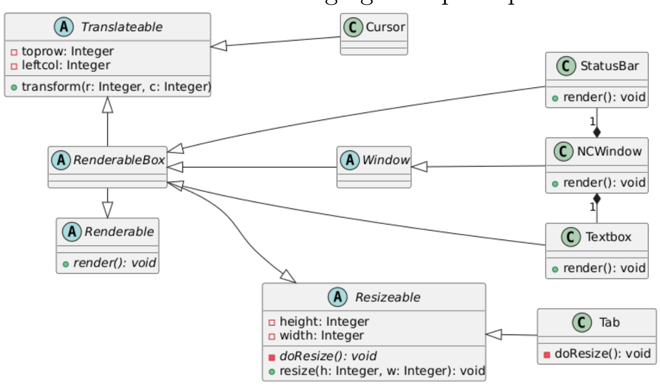


Figure 4. Above: view classes `NCWindow`, `Textbox`, `StatusBar`, as well as model classes `Tab`, `LinedFilebuf`, their managers, and `Window`. Below: separated interfaces of above classes in order to adhere to interface segregation principle.



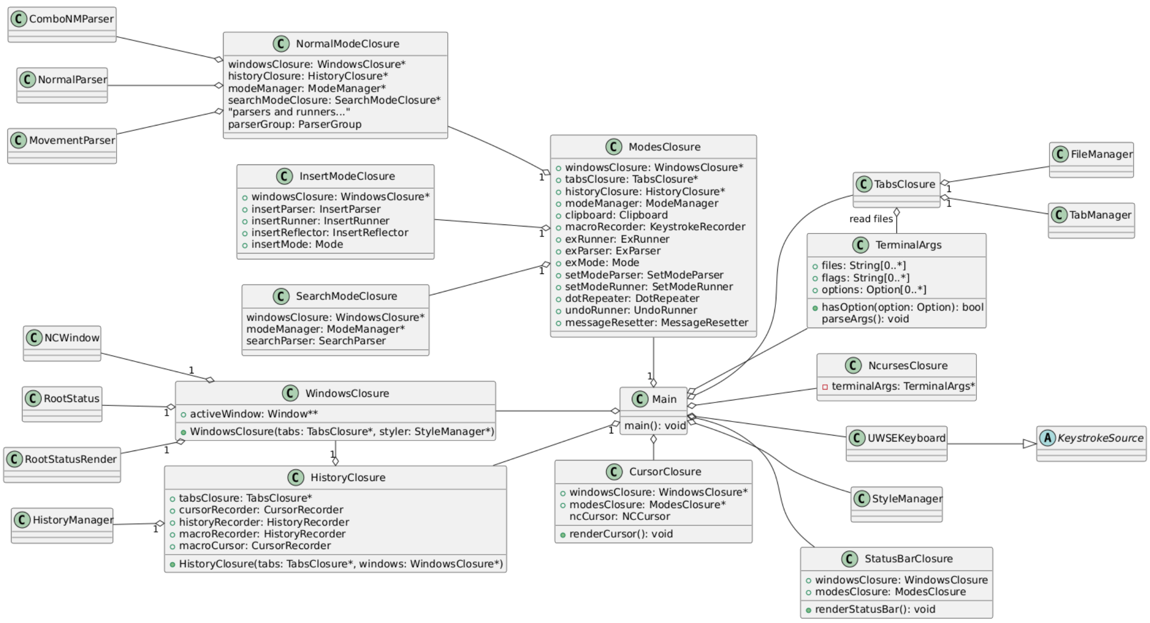


Figure 5. Classes used in `Main` to RAII and initialize and attach all parsers and runners.

3 Design

KeyStroke: following MVC, I abstracted user input using the `KeyStroke` struct, along with enum `Key` to store its type. This increases Cohesion as all user-input related classes can be put in one module, and the rest of the model doesn't need to deal directly with user input. The slightly increased coupling from other classes needing to read from `KeyStroke` subjects is reduced by CRTP.

Command: note that parsing and runnning commands should be separate. Hence, I added `Command` structs storing all info that could be parsed, and gave them parsers and runners. This increased cohesion by separating parsing from running, and commands from keystrokes. Note I decided on inheritance over variants, firstly because `Command`s differ a lot in size, so fixed-size variants are costly. More importantly, I can take advantage of covariant return types so that in the parsers, `Command* getCommand()` can be overriden to return `Normal* getCommand()`. This becomes important in the `CommandRunner` section, and wouldn't be possible with variants.

Observer Pattern: we skipped over Command pattern in lectures, so I improvised over observer pattern. I used this pattern twice: on `KeyStroke`s and on `Command`s. However, I differed from the textbook pattern in that I needed both to run without reading from its Subject: `ModeManager` should run `KeyStroke`s forwarded to it by `MacroRunner`, and all `CommandRunner`s need to run `Command`s forwarded by `DotRepeater` (the runner for normal-mode `.`) Furthermore, I noticed `notify()` methods mostly called `consume(subject.getKeystroke())` or `run(getCommand())`.
Hence, I used CRTP on `Subject<State,S>` and with `Observer<S>`, where `S` is the subject (for examples, refer to `lib/keystroke/keystroke_source.h`). Now, since `S` has a `getState()` method, we can specialize observers to call `getState()` in its notify, instead of passing a reference to a concrete `Subject` in each conrete `Observer`. Overriding `notify` is still useful. For example, in `InsertRunner`, since the user already typed a copy of the insert, in `notify()` we repeat `count-1` inserts, while `DotRepeater()` calls it with `count` inserts. This design decreases coupling between consumers and runners, as they don't depend on the concrete subjects anymore.

CommandRunner: I noticed most `CommandRunner`s were attached to one `Parser`, which only emits one type of `Command`, known at compile time. Hence, I templated runners and parser over the type of command they act on (e.g. `lib/command/runner/command_runner.h`). This way, the `run()` method can take as argument a derived command type, and `notify()` takes a `CommandSource<Type>`, whose `getCommand()` returns the derived type so that `dynamic_cast` is not required. Note that some runners (e.g. `JKRecorder`) should act on any type of command, hence I made `CommandSource<Command>` a base class of all other `CommandSources`. Covariant return types is necessary since we need to override getCommand in derived classes to return the derived type. This design

CommandParser: To parse normal mode commands, I needed parsers to stop parsing if the keystroke is invalid, and be reset when one of the parseres succeededs, or they all fail. Hence, I created a `ParserBase` class inheriting `Resetable` and `KeystrokeConsumer`. This way, I can create a `ParserGroup` and preform homogenous actions over a collection of `ParserBases`, e.g. checking if they are valid. Following interface templates, I made `consume() final` in the base and provided a private `parse()` method for subclasses to override. This way, parsers stop reading keystrokes when they fail.

Iterator Pattern: I had `LinedFilebuf` extend from a `LinedCharbuf`, which implements a bidirectional iterator, and a reverse iterator. The `begin()` and `rbegin()` for these take a line and column marking the start point of the iterator. This is useful for searching movements such as `w,b,f,/,?,%` and also helps with the `Textbox`, which uses the iterator to read and write chars. I decided to have iterators store line-col positions, which allows line-based arithmetic using pointers (useful for reading cursor position after searching). To support constant time lookup, `LinedCharbuf` was represented as a `deque` of `strings`, each terminated by a newline character. Hence, line-based operations like `dd` are constant time, and inserting within a line (i.e. insert mode) is linear in length of line (which is acceptable, as lines should be short). I noticed `line-col` pairs could be grouped into a `Loc` struct, and given an `operator<=>()` method to simplify comparisons (see `lib/matcher/matcher_runner.h`).

Strategy Pattern: to support search based movements, I wrote a `findNth` template over iterators (see `include/utility.h`) to return iterator to nth match within range of a `[beg,end)` pair of iterators, using a `Pred` (anything with `bool operator==(char)`) to find matches. I had the invariant that each character would be matched exactly once. Hence, implementing runners for search movements was: write a `Pred`, generate `[beg,end)` from the current cursor location, run `findNth`. Since iterators return their `Locs`, it remains to set the `Cursor` location into the `Tab`.
Some examples of `Preds` were `Chunks` (`lib/command/runner/movement_runner.h`), which I used to implement `w,b`. These store the previous character and uses it to check whether the current char marks the end/start of a word. By the invariant of `findNth`, each `operator==` would update prev. Hence, I used mix-in inheritance to so that `Chunk` updates prev. Hence inheriting from it and overriding `doCheck` suffices. This allowed me to easily create `Chunks` for `w,b,W,B`.

Word-Search: Another `Pred` was for the `?,/` commands, for which I implemented a matcher using KMP string matching algorithm. I had the constructor generate a Longest-prefix suffix (LPS) array from the search string that the user types into `Ex` mode, and move-assigned it to the matcher each time the user searches. Then, the `operator==` would match using the LPS. Reverse matching was implemented by constructing the LPS using the reversed string, and applying `findNth` using the reverse iterator obtained from `LinedCharbuf::rbegin()`.

Interface Segregation Principle: note that many of the operations on `Tabs`, `Windows`, `Cursors` are similar. For instance, `resize`, `translate`, and `render`. Hence, I captured the first two into concrete classes `Resizeable`, `Translateable`, and `render` into the abstract `Renderable`. Note that some classes such as `Windows` need to recalculate their contents on `resize`. Hence, I used **NVI** on `Resizeable`, providing a private virtual `doResize()` for subclasses such as `Tab` and `Window`.

Open closed principle: one specific instance I want to highlight which doesn't fit into existing patterns was the `Trie` class in `lib/matcher/matcher_trie.h`. I implemented it so that any class could extend the DFA by `add()`ing branches as `strings`. Although the extension still requires modifying the `Node::Type` enum, the logic for stepping through the DFA is constant. Compared to the `CommandParsers` I wrote earlier on, this design is much more extensible. For example, to add support for matching `/*` and `*/`, I added two `add(...)` statements to the constructor, two cases to the enum, and two `cases` in the `Runner`'s switch statement. Note that the enum could be extracted into a hierarchy (similar to `Command`) or another module, further increasing extensibility.

Prototype Pattern: notice that a command like `c2w` is a `SetMode` command, since it activates insert after deteting 2 words. Hence, I decided to separate c + Movement commands into `CM`, which inherits from `SetMode`. However, for `DotRepeater` to play back an `Insert` command, it needs to also replay the logic from its preceeding `SetMode`. Hence, `DotRepeater` needs to copy construct `SetMode`s for future use (note that parsers are always reset after notifying, so cannot shallow copy its pointer). Hence, I gave `SetMode` a `clone` method, so that `CM` could be copied dynamically.

Justifications: I justify implementing `Macro`s by recording and replaying keystrokes because that was easier than re-running a sequence of `Commands` (which would couple with all other runners). Instead, playing keystrokes into the the root `ModeManager` was simpler and more consistent. Furthermore, vim probably does this as well. Note that `:reg` shows recorded keystrokes (including `ctrl-G`), not commands. You can also change the keystroke recorded at a certain location. Hence vim macros are probably keystroke based.
I justify managing history in a `tree` and storing snapshots of the entire file since I think vim also does the same (although they probably store file-deltas, rather than whole snapshots). Vim undo branches can be accessed via `:undolist` and allows you to go to any edit made during the editing session. I wanted to implement the same, although I ran out of time. However, my `HistoryTree` class does store undo branchs, as evidenced through GDB (set a breakpoint on line 40 of `lib/history/history_tree.h`, watch `store`).
Why is `LinedCharbuf` concrete? This violates dependency inversion principle. But in this case, I think of `LinedCharbuf` as an implementation of in-memory line-based operations, which won't change much in the specification if at all.

4 Extra-credit features

- Multifile - turn on(off) by ex command `:multifile on(off)`. switch files with `:n`, `:N`. This was difficult to design, since multiple files means needing multiple cursors, window pane locations, etc. I solved this using the concept of a `Tab`, and used `TabManager` to handle file switches.
- Help: type `./vm -h` for a help message. I did this to show my extensions
- Colors: use `./vm --show-color --color-set standard` for colors. I did this because it is pretty, and also error codes (e.g. `:q` with unsaved changes) are highlighted
- Extended history: you can run `Ctrl+r` to redo commands. I added this since I use it frequently, and is a simple first step towards undo-branches that I wanted to add (see **Justifications**). I implemented this by storing undos into a "future stack", and popping from the stack to redo. Pushing a new change clears the future.
- No explicit memory management: I wanted this because it leads to cleaner, non-leaking code. I achieved this using non-owning pointers, `unique_ptr`s and `std` library collections
- Extended movements: `W`, `B`. These were extensions from `w,b`, all of which I frequently use. It also illustrates mix-in inheritance and strategy pattern well. See **Strategy Pattern**
- KMP pattern matching: allows for O(n+m) plain-text searching. See **Word-Search**

3 additional features I'm not certain are enhancements but would like to highlight: 1: Insert mode handles enter key (implemented because useful). 2: Normal mode handles arrow key movements, and mouse scrolling. Supporing these is why I added the `Key` enum, which made it easy to extend. 3: The `%` matcher reproduces behaviour for `#endif`, `#if`, `#elif`, `#else`, `/*`, `*/`. This was difficult in that it required the `Trie` data structure (see **Open closed principle**).

5 Final Question

Based on the `Trie` class I wrote for `%` runner, I would redesign the parsing of `Commands` so that each VM command's DFA path could be added via an `add` method, with exactly one `CommandRunner` corresponding to each `Command`. This would significantly decrease coupling, as each `Command` is isolated from others, and only commands such as `ComboNM` would rely on other Commands. This would

make general runners such as `DotRepeater` extremely repetitive, although this can be addressed using the Command pattern. That is, we make `Command::run()` a pure virtual method, and initialize each `Command` with a `CommandRunner`. Since there's a bijection between Commands and Runners, concrete `Command`s can have a `static` member runner. Then, `concrete Command`s override `run` to pass themselves into the runner's run, along with data such as `count`, `register`, etc. This way, `DotRepeater` can just call `run()`, as can `ComboNM` runners.
This was actually my original idea, however I could not figure out a reasonable way to represent a path, since one would need to parse `count`s, `register`s, and for `ComboNM`, multiple sub-DFA paths. Another disadvantage besides skill issue is that it would signficantly decrease cohesion, since many VM commands are extremely similar. However, coupling is a far larger issue in this context. First, the side-effects from observer-pattern combined with parser-groups resetting themselves already make it very hard to reason about how commands run. Adding more commands would be nearly impossible, and having people collaborate would require a very long learning period. Furthermore, extending parser functionality currently requires modifying parser classes (violating open-closed principle), whereas adding DFA paths leads to code that is cleaner, more efficient, and easier to reason about. Hence, I think the benefits outweight the duplicated code, and if the repetition is too bothersome, using mix-in inheritance can address the issue.