Project Overview
The objective of this project is to develop a Vim-like text editor using C++ and ncurses, guided by an object-oriented design that aligns with the provided UML specifications. The editor will feature both Insert and Command modes, enabling users to enter and manipulate text efficiently. Key functionalities include a wide array of commands for text manipulation, an unbounded undo system, syntax highlighting for C++ files, macro recording and playback, and a responsive user interface with a status bar. Additionally, the project aims to incorporate bonus features such as Visual Mode for enhanced text selection and manipulation.

Project Timeline
Day 1: Project Setup & Initialization
The initial phase focuses on establishing a solid foundation for the project. This involves creating the essential directory structure, including include/, src/, and build/ directories, and setting up crucial files such as CMakeLists.txt, which facilitates the build process when using CMake. Initial class headers like Editor.h and Display.h, along with their corresponding source files Editor.cpp and Display.cpp, are created to outline the core components of the editor. Basic classes for Editor and Display are implemented with minimal functionality to initialize and render an empty window using ncurses. Successful compilation and execution at this stage should result in an empty ncurses window with a status bar, verifying the setup's correctness.

Day 2: Mode Management & Basic Navigation Commands
The next phase introduces mode management and basic navigation. A Mode enumeration is defined to represent the two primary states of the editor: Command and Insert. The Editor class is enhanced to incorporate mode switching logic, allowing users to toggle between modes using the ESCAPE key. The current mode is prominently displayed in the status bar, providing clear feedback to the user. Basic navigation commands h, j, k, and l are developed to move the cursor left, down, up, and right, respectively. These commands interact with the IDocument interface to update the cursor's position, and changes are reflected in the Display. The Display contains a status bar to display critical information, including the current mode or file name on the left and row and column numbers on the right. Line numbering is implemented to aid in navigation and reference, and lines exceeding the window width are handled with proper wrapping to maintain readability. Additionally, the editor displays ~ symbols for empty lines beyond the document's end, providing a clear visual indicator of the file's boundaries.

Day 3: Text Insertion & Deletion, Command Execution Framework
Developing the ICommand interface for text manipulation and the command execution framework. Commands such as i (insert) and a (append) are implemented to allow users to enter Insert mode and add text seamlessly. Commands like x (delete character) and d (delete commands) are developed to enable text removal, providing users with essential editing capabilities. This will be done through InsertCommand, DeleteCommand and CommandParser class is created to interpret user inputs and instantiate the appropriate command objects based on the input received.

Day 4-5: Commands

Implementing commands like b, c, f, n, o, p, q, r, s, yy, y, A, F, I, J, N, O, P, R, S, X, ^, $, 0, ., ;, /, ?, %, @, ^b, ^d, ^f, ^g, and ^u. Additionally, support for numeric multipliers (e.g., 3j, 2dd) is integrated, allowing users to execute commands multiple times efficiently. The CommandParser is refined to ensure accurate parsing and execution of these complex commands, enhancing the editor's versatility and user control.

Day 6-7: Undo Functionality & Syntax Highlighting
The u (undo) command is implemented with an unbounded undo stack, allowing users to revert an unlimited number of previous actions. Syntax highlighting to improve code readability, particularly for C++ files (.h, .cc). The ISyntaxHighlighter interface and its concrete SyntaxHighlighter class are crafted to define and apply syntax rules covering keywords, numeric literals, string literals, identifiers, comments, and preprocessor directives. It will also detect mismatched braces, brackets, and parentheses.

Day 8-9: Colon Commands & Search Functionality
Implementing colon commands such as :w (write/save), :q (quit), :wq (write and quit), :q! (force quit without saving), :r (read), :0 (jump to first line), :$ (jump to last line), and :line-number (jump to specified line) are developed. A CommandLine component is created to capture and display these commands at the bottom of the screen, executing them upon the user pressing Enter. Search functionalities using / and ? are implemented to allow users to search for exact string matches within the document. Search results are highlighted, and users can navigate between matches using n (next) and N (previous).

Day 10-11: Macro Recording & Playback
Introducing macro functionality and the implementation of advanced commands. The IMacroRecorder interface and its concrete MacroRecorder class are developed to facilitate the recording and playback of macros. Users can initiate macro recording using q followed by a macro name (e.g., qa to record macro a) and replay macros using @ followed by the macro name (e.g., @a). The system ensures that only recordable commands are captured within macros, excluding actions like cursor movements to maintain macro integrity.

Day 12: Visual Mode
implementing bonus features and optimizing the editor's performance. Visual Mode is introduced, encompassing Character Mode (v), Line Mode (V), and Block Mode (Ctrl+v), allowing users to select and manipulate text more intuitively. During selections, the status bar updates to display VISUAL mode, and standard commands (d, y, c, etc.) are enabled to operate on the selected text. Dynamic feature toggles are implemented, allowing users to enable or disable enhancements via command-line flags (e.g., --enable-visual) or runtime commands (e.g., :toggle visual). This flexibility enhances the editor's adaptability to different user preferences and requirements.

Day 13: Testing

Comprehensive testing and debugging are conducted to ensure all components function harmoniously. Unit tests are performed on individual modules, while integration testing verifies the seamless interaction between components. User acceptance testing is carried out to validate the editor's usability and functionality from an end-user perspective. Identified issues are debugged and resolved to enhance the editor's stability and reliability.

Day 14: Documentation
Finally, thorough documentation is compiled, including a user manual detailing usage instructions and command references, and developer guidelines outlining class descriptions and code structures. The design document is finalized, incorporating this "Plan of Attack" to serve as a comprehensive guide for the project.

Conclusion
This "Plan of Attack" outlines a structured and methodical approach to developing a feature-rich Vim-like text editor in C++ with ncurses. By adhering to this plan, the project ensures the systematic implementation of required functionalities, alignment with object-oriented design principles, and timely completion within the designated timeline. Continuous testing and a flexible contingency plan further enhance the project's reliability and adaptability, paving the way for a successful and robust text editor.