Matthew Rumbel
CSE 3502
C++ Essentials

What is this project about?

This project's main idea was to give students' C++ programming practice by writing a console application: a vi-like text editor.

Main Functionalities

- Normal Text Editing
    - Users can insert/remove characters similar to how other popular text editors work.
- Undo/Redo
    - Users can undo/redo text insertion/deletion as they see fit, similar to how other popular text editors handle undo/redo.
- Reading a File
    - Users can specify a normal .txt file to read as input to the editor
- Saving
    - As a user continues to work on the document, the program will automatically save the user's work
- Page Views
    - The editor will only show the amount of text that can fit on the screen, rather than a typical scrolling effect. For example, if the screen can only hold 3 lines of text, but there are 4 lines in the file, the lines to be displayed on the screen will vary depending on cursor location
- Line Wrapping
    - Similar to how MS Word follows word/line wrapping, this text editor is designed to support multiple formatting outputs (currently, sublime-style and word-style formatting)

CSE 3150 - Basic Text Editor Manual

By Matthew Rumbel

Overview:

This project will allow me to practice C++ programming by writing a console application: a vi-like text editor. Throughout this project, I will learn and gain experience with using different object-oriented approaches, such as Model View Controller and Observer pattern styles of design. Although the text editor itself is not particularly exciting to implement, the point of this project is to gain experience implementing a non-trivial program following the OO paradigm.

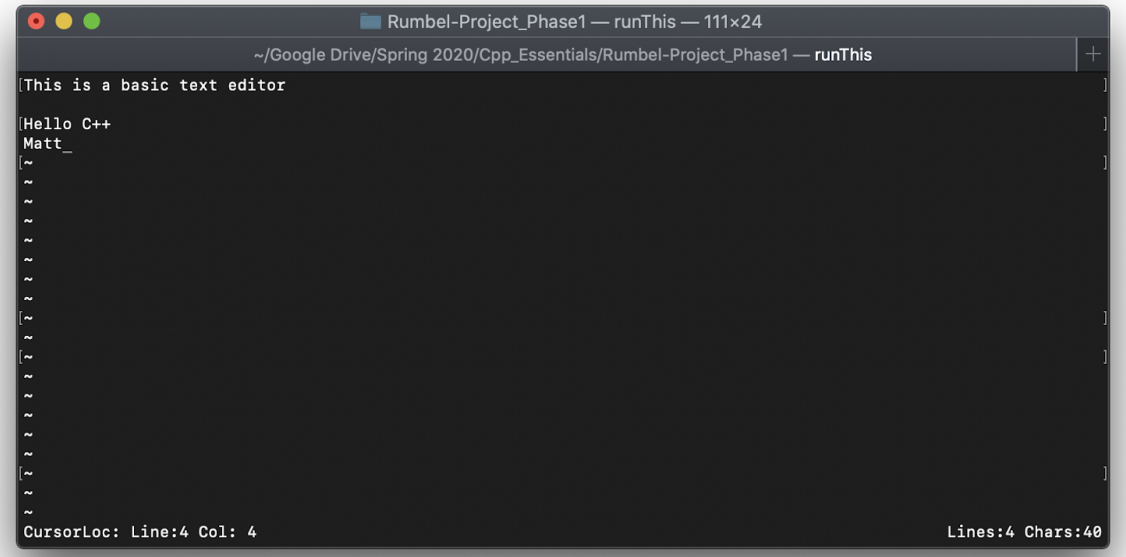
Table of Contents

Compile and Run

To compile this and run this program, please use the *make* utility on a Unix-based system. Using the included Makefile, the *make* utility will compile the necessary files and create an executable titled 'runThis'. A user can run the command **./runThis** to start the program.

```
$ cd Rumbel-Project_Phase1
$ make clean
$ make
g++ -c Commands.cpp -o Commands.o -O3 -std=c++14 -Wall -I.
g++ -c Document.cpp -o Document.o -O3 -std=c++14 -Wall -I.
g++ -c ECEditorTest.cpp -o ECEditorTest.o -O3 -std=c++14 -Wall -I.
g++ -c ECTextViewImp.cpp -o ECTextViewImp.o -O3 -std=c++14 -Wall -I.
g++ -c EditorController.cpp -o EditorController.o -O3 -std=c++14 -Wall -I.
g++ -o runThis Commands.o Document.o ECEditorTest.o ECTextViewImp.o EditorController.o
$ ./runThis <filename>
```

After the last command is run, the text editor will open and will be ready for text manipulation.



Features

In this current version of text editor, functionality for basic text editing features are available for use. These include, inserting characters, carriage return, undo, and several others. Here is a list of all available features:

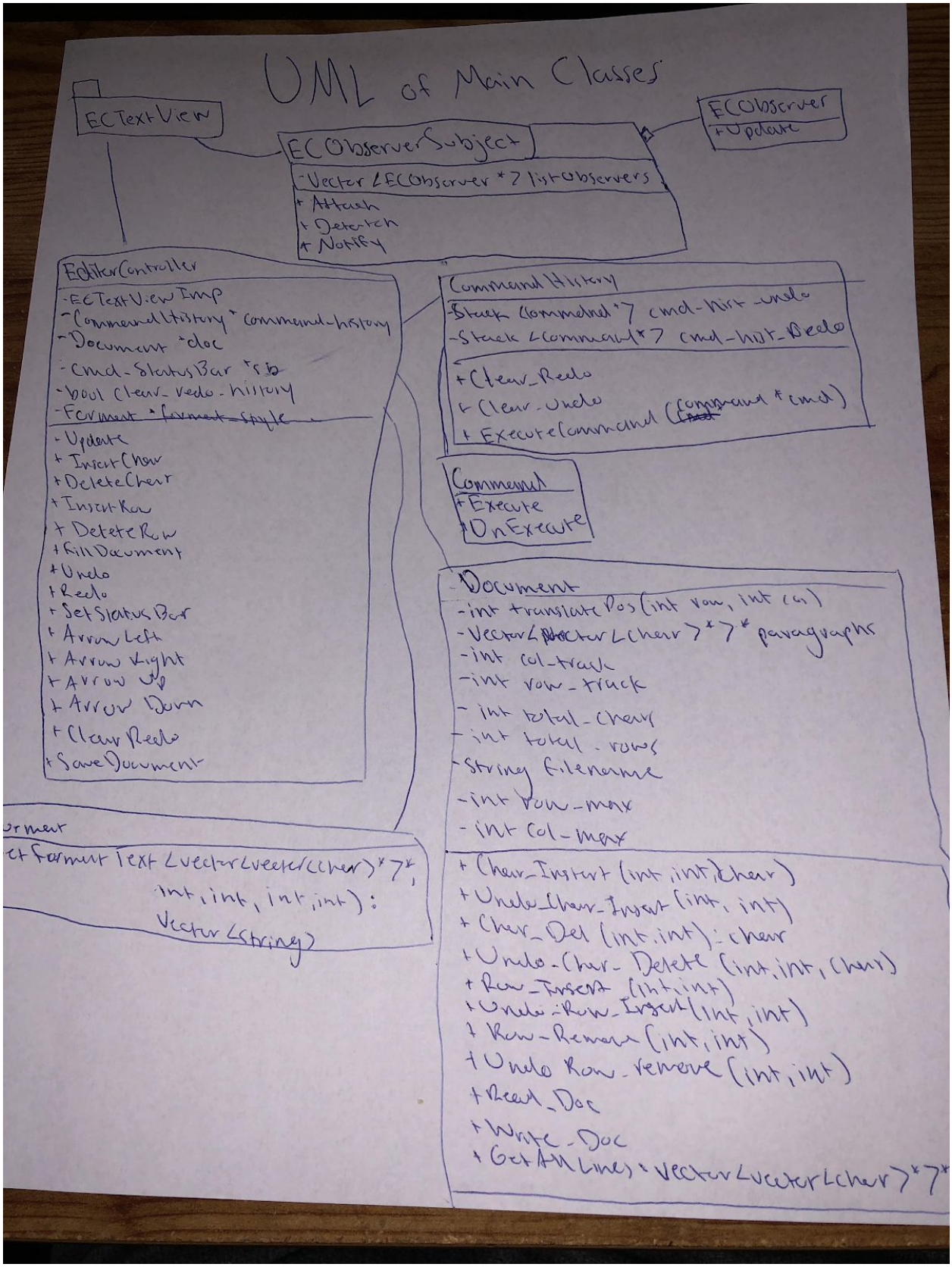
- Text Insertion
    - A /symbol can be inserted into the editor by pressing a desired character key from the keyboard (for example, pressing the letter **a**, number **7**, or symbol **+**.
- Text Deletion
    - A piece of text can be removed from the editor by pressing the **backspace** key (or **delete** key on MacOS). This will remove the symbol at the cursor position, if there is one to remove. If the cursor is all the way to the left of a row of text, the text deletion feature will move the text on the current row to the end of the previous row (if on row > 0).
- Line Break
    - A user can insert a line break by moving the cursor to the desired location and then tapping the **enter** key (or **return** on MacOS). On the same line, if there is text after the cursor location, that text will be moved to the new row.
- Undo/Redo
    - In this implementation, Undo and Redo will behave in a similar manner as other common text editors. As a note, you cannot Redo a command (text insertion, deletion, etc…) that has not been Undone. A user can press **CTRL-Z** to Undo and **CTRL-Y** to Redo.
- Quit
    - A user can quit out of the text editor by pressing **control** and **Q** (**Ctrl + Q**) keys at the same time. This will terminate the program.
- Specific File Editing
    - If you would like to edit a certain basic text file, include the file name as a command line argument when executing the *runThis* executable. See the **Compile and Run** section of this document for specific syntax. While editing, the file will auto-save, no need to press a unique button to tell the system to save your file.

User Interface

In this implementation, once text insertion has begun, the user will see a status bar located at the bottom of the text editor. The status bar will show useful information such as total lines and total characters within the editor, as well as current location.

Feature Overview

Feature	Status	Design Pattern	User Manual
Text Editing/Cursor Movement	Implemented	• Observer • Model-View-Controller • Composition	Keyboard entry for all letters supported
Undo/Redo	Implemented	• Model-View-Controller	All commands can be undone/redone
File Load	Implemented/ Some bugs depending on platform		To see specific syntax, see above manual
File Save	Implemented/ Some bugs depending on platform		To see specific syntax, see above manual
Output Formatting	Bugs	• Strategy	By default, format style is set to Sublime Text app style formatting. Word document editor-style formatting coming soon


UML of Main Classes
ECTextView
ECObserverSubject
-Vector<ECObserver*> listObservers
+Attach
+Detatch
+Notify
ECObserver
+Update
EditorController
-ECTextViewImp
-CommandHistory* command_history
-Document *doc
-Cmd_StatusBar *sb
-bool Clear_redo_history
-Format *format_style
-Update
+InsertChar
+DeleteChar
+InsertRow
+DeleteRow
+FillDocument
+Undo
+Redo
+SetStatusBar
+ArrowLeft
+ArrowRight
+ArrowUp
+ArrowDown
+ClearRedo
+SaveDocument
CommandHistory
-Stack<Command*> cmd_hist_undo
-Stack<Command*> cmd_hist_Redo
+Clear_Redo
+Clear_Undo
+ExecuteCommand(Command *cmd)
Command
+Execute
+UnExecute
Document
-int translatePos(int row, int col)
-Vector<Vector<char>*>* paragraphs
-int col_track
-int row_track
-int total_chars
-int total_rows
-string filename
-int row_max
-int col_max
+Char_Insert(int,int,char)
+Undo_Char_Insert(int,int)
+Char_Del(int,int): char
+Undo_Char_Delete(int,int,char)
+Row_Insert(int,int)
+Undo_Row_Insert(int,int)
+Row_Remove(int,int)
+Undo Row_remove(int,int)
+Read_Doc
+Write_Doc
+GetAllLines: Vector<Vector<char>*>*
+SetFormatText<Vector<Vector<char>*>*, int,int,int,int): Vector<string>

Class (Selected)	Key Method(s)
ECObserverSubject	● Attach (Add an ECObserver) ● Detach (remove an ECObserver) ● Notify (notify ECObserver(s) that a change has been made)
EditorController	● Update (update the document) ● Depending on the update, other methods called
CommandHistory	● Redo (Redo an executed command) ● Undo (Undo an executed command)
Command	● Execute (do something to the document) ● UnExecute (undo something done on the document)
Document	● Methods handling the actual manipulation of text/cursor and document reading/saving
Format	● Formats the text that should be shown to the screen

Key Technical Issues

- Cross Platform
    - No platform-specific functionalities could be added
- Cursor Manipulation
    - Since the text displayed on the screen may not (all) be the text in the actual document, time was spent figuring out a way to translate a position on the screen to a position in the document itself

Key Algorithms

- Inserting/Removing Text/New Lines
    - Since each of these commands manipulate the text, a chain of things needed be done, including:
        - Actual text manipulation
        - Cursor manipulation in a way that is intuitive
- Paragraph Formatting
    - Since different styles of formatting may be desired, the program is designed to abstract formatting the text on the screen, given the text in the document. Things like word/line wrapping needed to be considered depending on the format type.

Refactoring

- Different OO Principles were applied throughout this project
    - Observer Pattern
        - Corresponds to *who* should be making changes when a command is started
    - Model View Controller Pattern
        - Having a main controller separate from the document allows for an abstraction between the document being written, formatting to the screen, and separation of commands
    - Composite Pattern
        - Document can be considered a composite of objects that hold text (vectors)

What I Have Learned

This project has taught me a lot about different software design principles. The first thing that was unique about this project was the scope. Since the editor was a long term project with lots of features/functionalities, it was important to make smart decisions with regards to project design. This text editor project was the first time I had to go in depth with smart design patterns (mentioned in the Design Specifications document). Combining many different class and objects together into something cohesive and intuitive was not easy, and really helped me gain experience with large projects, dynamic requirements, and long-term goals.

Future Improvements

The main goal for the future would be to improve efficiency of the program using smarter design choices. As a simple example, the program efficiency may be improved by only saving the document when the user wants (or allow the user to specify how often the document work should be saved). Additionally, the intuitiveness of the overall project design may be increased if more base objects are implemented. As a simple example, paragraph or line objects rather than a combination of vectors holding characters.