***I.T. Project Documentation***

***By Bradyn Walsh***

**Introduction**

My project, the Xi-Win Editor, is a Windows/.NET frontend for the Xi-editor text editor being developed as part of a Googler’s 10% Free Project Time.

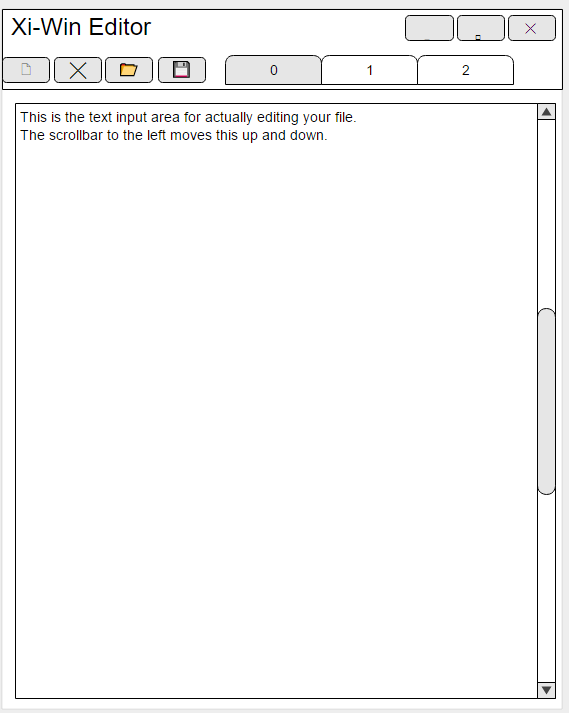
**Purpose**

The purpose of my project is to create a text editor that uses the xi-editor core that is currently in a very early development stage for all text manipulation operations. This text editor should be intuitive and easy for users to use. It will allow for having multiple tabs open at once using tabs and will be able to open and save files per user input. Additionally, it will have a command mode like Vim’s (another text editor) for keyboard shortcuts to make activating them easier for people with disabilities.

**Innovation**

This text editor differs from a standard text editor because it uses an external process (the xi-editor core written in Rust) to handle all text manipulation. This means that the developer of a frontend for the core, which is what my project is, does not have to focus on writing high-performance text manipulation operations and can instead focus on building a native interface for the frontend. Additionally, it will be easy to extend and improve my text editor as more work is done on the core as it is a whole lot easier to add a few commands to communicate to the core with than to just write them how you would for a normal text editor.

**Mock-up of User Interface**



**Design Decisions**

As you may notice in the UI mock-up, the tabs are numbered rather than given a proper name, like the file they are editing. This is a current limitation of the core process, but when it is fixed in the core it will be automatically fixed in the text editor. But while the tabs take up little space, I decided to put the buttons for creating, closing, opening and saving on the same line as the tab bar to save some vertical space which can then be used for showing more text.

Another design decision made was to have the text editor open its README.md file whenever it launches, which cannot be seen in the mock-up. This is very like what Vim and Emacs do on launch and should be helpful to any people new to using the editor as it explains how to use it and current issues with the editor. Additionally, advanced users can use this feature to quick-open any file on launch by replacing the README.md file, which is quite a useful feature to have in some circumstances.

**Target Audience**

The target audience for my text editor is people who use text editors everyday as part of their professional life, such as developers. This is because this app can be customized extensively by adding plugins to the core process, meaning that professionals can optimize their day-to-day workflow.

**Consistency**

My text editor is quite consistent in its rather Spartan interface. All the buttons for interacting with the editor use Unicode icons instead of text, all UI elements that are next to each other or otherwise related have the same size and colour scheme is consistent with standard Windows theming throughout the whole application.

**Help**

Whenever someone launches my text editor, it shows up a help file. This is extremely useful to users who have never used it before as it quickly runs through how to use it, how the command mode and keyboard shortcuts work and current bugs and issues with the editor. Additionally, as this project is hosted on GitHub, any user can submit an issue to the main repository if they have a problem and I can see if I can fix it.

**Accessibility**

My text editor is very accessible and intuitive to use for anybody. Using icons for the buttons means that non-English speakers can still use the editor and as the editor does support Unicode characters because of accessibility, they can input non-English characters. Additionally, many of the UI elements, such as the tab bar, can be found in many other applications and therefore should be very easy and intuitive for any user to use.

If I could add more accessibility features, I would add the ability to change the font size and colour and the colours for the rest of the UI, which would help visually impaired and colour blind people.

**Modules and Strategies for Development**

My text editor can be very easily broken down into 3 distinct and separate modules:

1. ***CoreCommunication Module***

This module of the text editor is responsible for starting up and communicating with the xi-core process. It is also responsible for closing the core process and cleaning up after it when the user closes the program.

To develop this, it would probably be best to put this module in its own class and use C#’s Process and Thread libraries for the heavy lifting of communicating with and starting/closing the core.

1. ***Command Interface Module***

To make it a lot easier to develop the frontend, it would be good to have a command interface we could hook into to parse incoming commands and to compose the JSON we need to send from standard C# objects. This would mean that our frontend will not have to deal with raw JSON strings but rather with nicely typed objects.

To develop this, I think that it would be best to create an interface describing what a command can do (be converted to JSON, be parsed from a string, etc.) and then create a class for each type of command implementing this interface.

1. ***Editor Frontend/UI Module***

This module is responsible for displaying the UI, processing incoming commands into text and sending out user keypresses for the core to input as text. Additionally, this module needs to handle opening tabs, closing them and file manipulation (open/save) through the commands provided by the core.

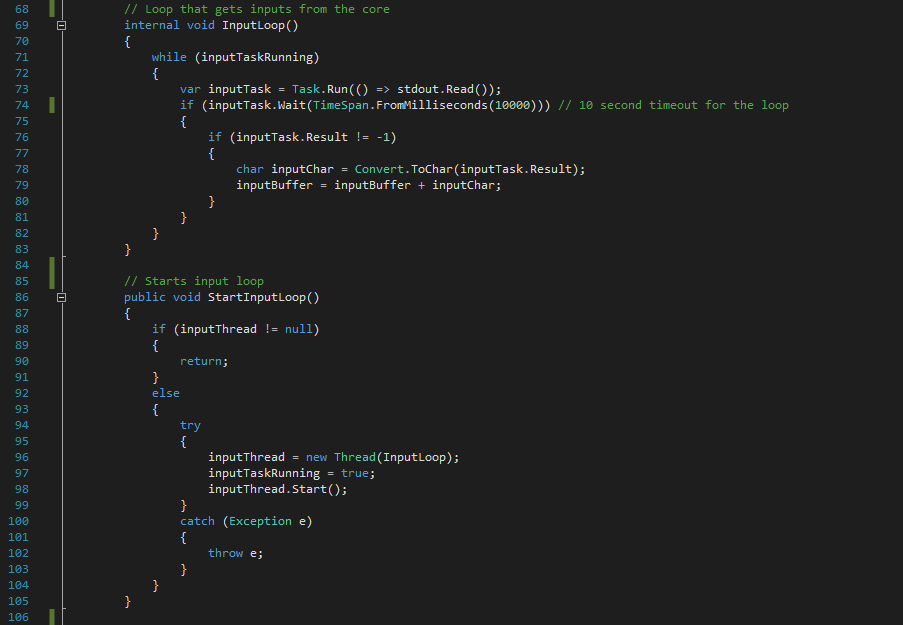
To develop this, it would be best to use the XAML designer to design the interface and then put all user interaction code into the resulting .cs file. This code should be extensively using the other 2 modules for doing all the heavy lifting and it should just focus on updating the UI elements and processing user input.

Here is what would happen when the user presses the new tab button:

C:\Users\walshb\Downloads\Untitled Diagram.png

**Explanation of Coding**

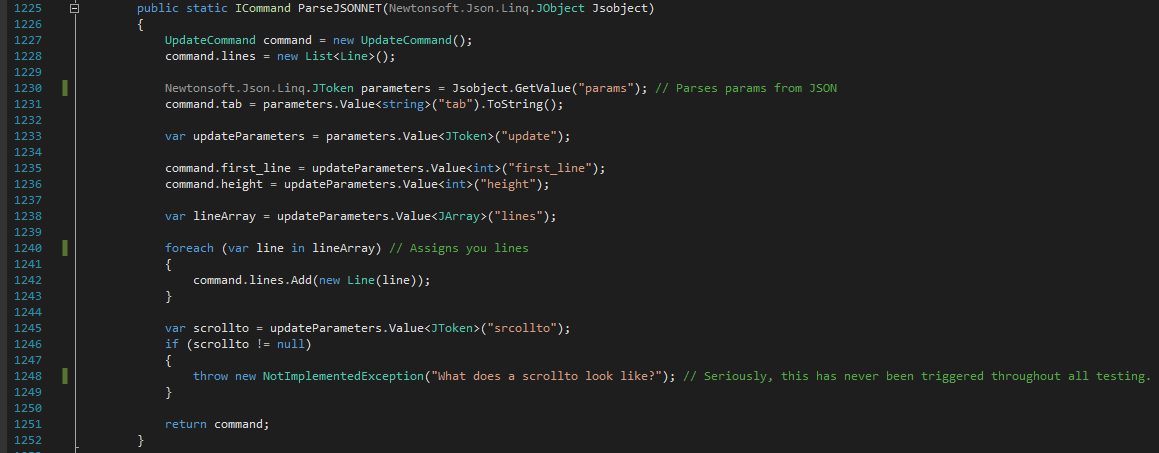
***Core Input Processing***



A major problem I had while developing the application was the application stalling because the app was sending info to the core process and waiting for it to be read while the core was writing something to my program waiting for it to be read. To fix this deadlock, StartInputLoop creates a new Thread which runs the InputLoop function.

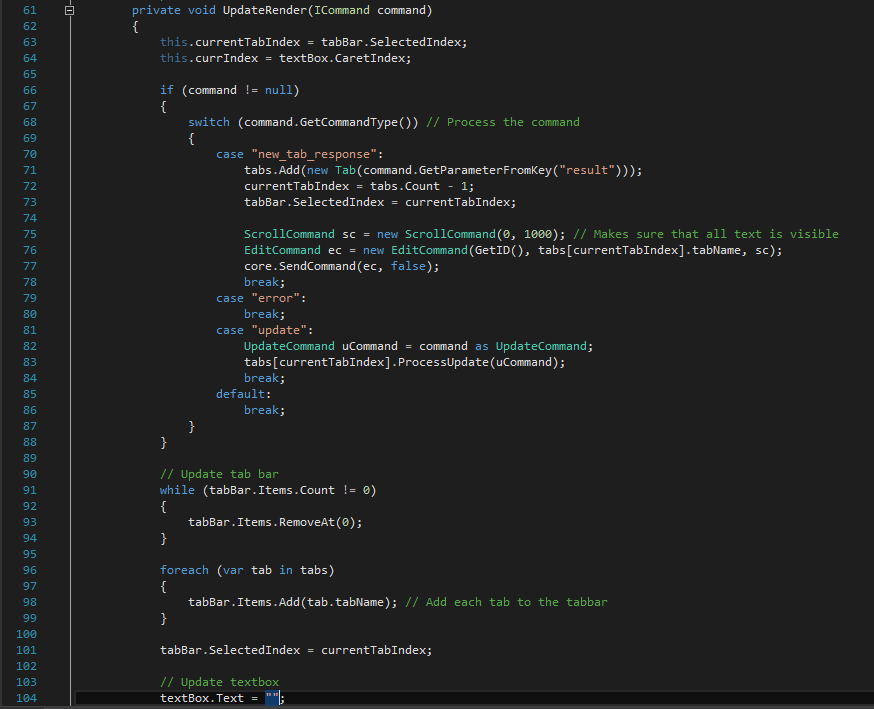
The InputLoop function simply tries to read from the core process, timing out and restarting every 10 seconds. If it does receive any input, it will add it to the inputBuffer which is a string used by the rest of the CoreCommunication class. It will continue to do this as long as inputTaskRunning, which is a Boolean defined in the class, is true.

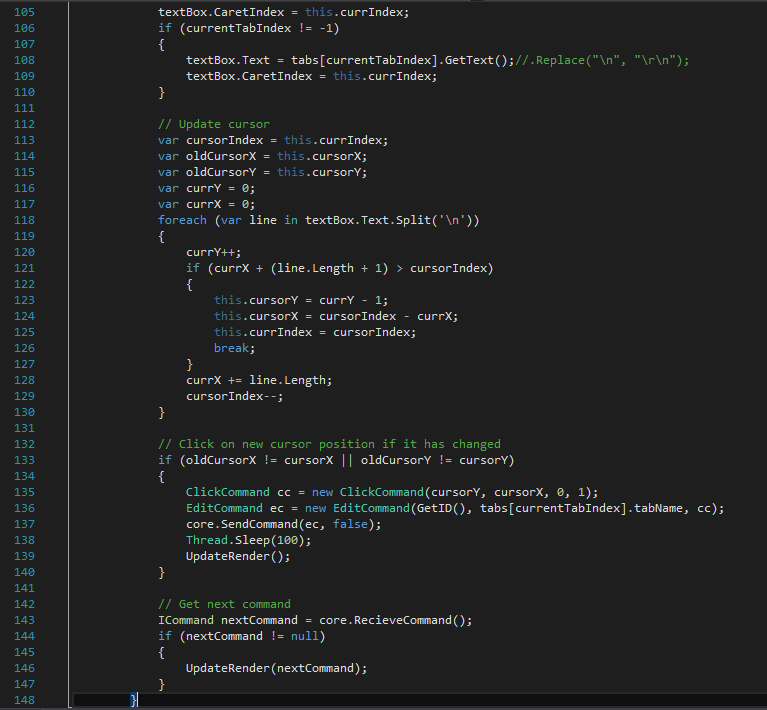
***Update Command Parsing***



This function, which is in the UpdateCommand class, is responsible for parsing all ‘update’ commands that the core sends to our application. It starts by creating a black updateCommand and initializing the lines array. It then extracts the parameters from the JSON using Newtonsoft’s JSON library. From these parameters, it extracts all the values inside and puts them in the correct variable in the class. It also then parses each line given by the core into a Line object. This function does throw an exception if the core returns a ‘scrollto’ value, but I have not seen that happen with the commands currently implemented.

***Updating and Rendering UI State***





This function is responsible for processing all incoming commands and updating the state of the UI. The first if and switch statement process all the incoming commands, depending on their type. Then we remove all items from and repopulate the tab bar with all the tabs that we have stored. After that, we clear the text input box and insert what should be there, based off what tab is open. Then we figure out the 2D coordinates of the cursor, which is what the core uses, from the 1D coordinates provided by C# and send a click to the core if the caret (text cursor) has moved. After that we see if there is another command. If there is, we call the function again with that command and if not, we just return.