CMP2092M Programming Paradigms Report

When deciding to create a piece of software, several factors need to be taken into consideration including what type of language to choose, all languages fall into one of several programming paradigms depending on their functionality. The four main paradigms are Procedural, Logical, Functional and Object-Oriented. For this project a text editor should be made to manipulate text that has been inputted from a file location, this manipulation includes:

- moving a cursor either over one character or to the start/end of the sentence string
- highlighting parts of the text input, be that a single character, a word or the whole of the sentence on either side of the cursor
- being to cut, copy and paste the highlighted words
- being able to perform a backspace or delete
- Once completed the text should be able to be outputted to a file location also

Shown below are the pros and cons of each paradigm in the context of the project followed by which should be suggested for the creation of the software.

Procedural:
A procedural language consists of a series of step by step coding that is compiled linearly. Coding in this language is fairly simple and can interpreted by compilers easily, it is also can be efficiently structured for quick editing in smaller projects. This is countered by the fact that security is usually fairly minimal, this is especially troubling for use in this project as the main aim is to create the software for a cryptography company who wishes to make it more secure than 3rd party software. The operations performed are also given more importance than the actual data which is problematic given the context of the project. Lastly its lack of modularity can become tedious when large amounts of code are implemented, luckily in this case the coding for the text editor would be reasonably lightweight but if features were added later on it could get to the point where it becomes incredibly difficult to edit.

Object-Oriented:
Classes and Objects are a key part to an Object-Oriented language, each object is created with its own data structure and so can easily be related to the real world if a project calls for it. The modular nature of the paradigm means that maintaining and improving an established piece of code is simple and effective. The main issue for these languages is that simulating real world objects can be hard if they do not fit neatly into distinct categories, this isn't a big issue for this project however. The main problem with this paradigm for use in the creation of a text editor is that when using if statements an object can interact in unexpected ways, this is trouble because the company who have given the project did so to ensure they could keep a tight grip on what their program is doing. Extra resources could be given to ensure the code written can not act in any way deemed problematic but there is always the possibility that this issue will occur.

Functional:
Functional programming involves using formal methods to produce programming solution, focusing on using mathematics in computation. Functional languages usually provide a safe environment to code in and allows for programmers to make quick prototypes, it is a useful paradigm for initial development but is not designed to fit hardware and so may not be a good choice for a final product.

Logical:
logical languages are based on mathematical logic. They are useful for complex processes because the tedious coding is completed by the engine running the language. Logical programming is very inefficient however and would not be as lightweight as the company wishes, it also is only useful in very niche applications and a text editor may not be one of them.

Comparison and Conclusion:
After looking at all four of the main paradigms individually it is time to compare them to each other, while procedural is a simple language and the project is to create a simple text editor so they would work well together, there is very little in the way of data security which is exactly the opposite of what the company wants, this is in comparison to object-oriented which can be given security in a number of ways but it also can have code interacting in ways that were not thought of. Functional is very good earlier in the code's creation, but not as useful later on, Logical however may not be useful at all for the given project so it may be wise to choose one that works well at some stages rather than not at all. Object-oriented allows for speedier upgrading in the future after development has finished and so could be seen as better than functional which has trouble working with hardware and is designed to be used in the earlier stages.

If given the choice it seems that using the functional paradigm to develop the prototype and give the company something they can observe before finalisation on code begins would be a good step. Once that was done object-

oriented would be a good choice for the final implementation to allow for the most versatility and functionality while keeping security as high as possible.

References:

https://neonbrand.com/website-design/procedural-programming-vs-object-oriented-programming-a-review/
https://www.codementor.io/learn-programming/comparing-programming-paradigms-procedural-programming-vs-object-oriented-programming
https://www.quora.com/How-can-you-compare-different-programming-paradigms-in-laymans-terms
https://shunt00213.blogspot.co.uk/2010/09/project-4-advantages-and-disadvantages.html

Programming Paradigms ADT

NAME

Text Editor – an ADT representing text editing software

SETS

T	the set of text editing (S x S x S x S)
L	the set of loading (S)
N	the set of Natural numbers ($\forall$ n.n $\in$ Z $\wedge$ n $\geq$ 0)
C	the set of Characters ($\forall$ c.c $\in$ [a..z])
S	the set of Strings (N x C)
H	the set of highlighting (S)

SYNTAX

create:	$\perp$	→	T
destroy:	T	→	$\perp$
init:	T	→	T
setFile:	T	→	L
getFile:	L	→	T
move	T x C x N	→	T
concatenate:	S x S	→	S
highlight:	T x C x N	→	H
cut :	T x H	→	T
copy :	T x H	→	T
paste :	T x H	→	T
remove:	T x C x H	→	T

SEMANTICS

$\forall$t.t$\in$T, $\forall$b.b$\in$L, $\forall$l.l$\in$S, $\forall$r.r$\in$S, $\forall$h.h$\in$H

pre-create():: true

post-create(t):: t = (“”, “”, “”, “”)

pre-destroy(t):: true

post-destroy(t):: t = $\perp$

pre-init():: true

post-init((_,_,_,_);t):: t = (“”, ”””, ”””, ”””)

pre-setFile():: t = (“”, ””, ””, ””)

post-setFile(t, b):: b = t

pre-getFile():: true

post-getFile(t, b):: t = b

pre-move(t):: t = (l, ””, r, ””)

post-move(t):: t = (“”, l, r, “”)

pre-concatenate(t, l, r):: t = (l, ””, r, ””)

post-concatenate(t, l, r):: t = (l, r)

pre-highlight(t, l, r):: t = (l, “”, r, “”)

post-highlight(t, h):: t = (“”, “”, “”, h)

pre-cut(t, h):: t = (l, ””, r, h)

post-cut(t, h):: t = (l - h, ””, ””, h)

pre-copy(t, l, r, h):: t = (l, ””, r, h)

post-copy(t, l, r, h):: t = (l, ””, r, h)

pre-paste(t, l, r, h):: t = (l, ””, r, h)

post-paste(t, l, r, h):: t = (l + h, ””, r, h)

pre-remove(t, l, r):: t = (l, ””, r, “”)

post-remove():: t = (l - 1, ””, r, “”)

```
--module Main where

import Data.List
import Data.Char
import System.IO
import qualified Data.Char as Char --Used to convert highlighted text to Capital

data TxtEdit = TxtEdit {l :: String, c:: String, r:: String, h::String} deriving (Show)
-- leftSentence,  cursorPosition, rightSentence, highlight_Editing
data ErrorChecking = Testing {a :: Int, s :: Int} deriving (Show)
-- cursorLocation, sentenceLength
--data FilePath
--setup :: TxtEdit -> TxtEdit -- Setup for further coding
--setup(TxtEdit{l = ls, r = rs, c = cl, s = sl, h = he}) = TxtEdit{l = "", r = getLine, c = 0, s = getline.Length, h = he}

initiate :: TxtEdit -- Initiate Text
initiate = (TxtEdit {l = "Text Editor",c = "|", r = "Test", h =""})

--init2 :: String --Test String initializing
--init2 = (conccurs(TxtEdit{l = "Text Editor",c = "|", r = " Test", h =""}))

--i3 :: String -> TxtEdit
--i3 ("Text Editor", "|", "Test", "") = (TxtEdit{l, c, r, h})

moveLeft  :: TxtEdit -> TxtEdit -- Move Cursor Left
moveLeft  (TxtEdit{l = ls, c = cp, r = rs, h = he}) = (TxtEdit{l = (reverse(tail (reverse ls))), c = cp,  r = [(head
(reverse ls))] ++ rs, h = he})

moveRight :: TxtEdit -> TxtEdit -- Move Cursor Right
moveRight (TxtEdit{l = ls, c = cp, r = rs, h = he}) = (TxtEdit{l = [(head(reverse rs))] ++ ls, c = cp,  r = (reverse(tail
(reverse rs))), h = he})

moveStart :: TxtEdit -> TxtEdit -- Move Cursor to the start
moveStart (TxtEdit{l = ls, c = cp, r = rs, h = he}) = (TxtEdit{l = "", c = cp, r = ls ++ rs, h = he})

moveEnd  :: TxtEdit -> TxtEdit -- Move Cursor to the end
moveEnd  (TxtEdit{l = ls, c = cp, r = rs, h = he}) = (TxtEdit{l = ls ++ rs, c = cp, r = "", h = he})

concatenate :: TxtEdit -> String -- Show Entire Sentence, concatenate left and right sentences into one
concatenate (TxtEdit{l = ls, r = rs}) = "Entire String: " ++ ls ++ rs

conccurs :: TxtEdit -> String -- Concatenate with cursor, will be used to possibly abstract code from user
conccurs (TxtEdit{l = ls, c = cp, r = rs }) = ls ++ cp ++ rs

charHighLeft :: TxtEdit -> TxtEdit --Highlight character left of the cursor
charHighLeft (TxtEdit{l = ls, c = cp, r = rs, h = he}) = (TxtEdit{l = (reverse(tail (reverse ls))), c = cp, r = rs, h =
[(head (reverse ls))] })

charHighRight :: TxtEdit -> TxtEdit --Highlight character right of the cursor
charHighRight (TxtEdit{l = ls, c = cp, r = rs, h = he}) = (TxtEdit{l = ls, c = cp, r = (reverse(tail (reverse rs))) , h =
[(head (reverse rs))] })

strHighLeft :: TxtEdit -> TxtEdit --Highlight string left of the cursor
strHighLeft (TxtEdit{l = ls, c = cp, r = rs, h = he}) =(TxtEdit{l = "", c = cp, r = rs, h = ls })

strHighRight :: TxtEdit -> TxtEdit --Highlight string right of the cursor
strHighRight (TxtEdit{l = ls, c = cp, r = rs, h = he}) =(TxtEdit{l = ls, c = cp, r = "", h = rs })

wordHighLeft :: TxtEdit -> TxtEdit --Highlight word left of the cursor
wordHighLeft (TxtEdit{l = [], c = cp, r = rs, h = he}) = (TxtEdit{l = [], c = cp, r = rs, h = he})
wordHighLeft (TxtEdit{l = ls, c = cp, r = rs, h = he}) = if ([head (reverse ls)]) == " "
```

```
then (TxtEdit{l = ls, c = cp, r = rs, h = he})
else wordHighLeft (TxtEdit{l = (reverse(tail (reverse ls))), c = cp, r = rs, h = [(head (reverse ls))] ++ he})

wordHighRight :: TxtEdit -> TxtEdit --Highlight word right of the cursor
wordHighRight (TxtEdit{l = ls, c = cp, r = [], h = he}) = (TxtEdit{l = ls, c = cp, r = [], h = he})
wordHighRight (TxtEdit{l = ls, c = cp, r = rs, h = he}) = if ([head rs]) == " "
then (TxtEdit{l = ls, c = cp, r = rs, h = he})
else wordHighRight (TxtEdit{l = ls, c = cp, r = (tail rs), h = he ++ [head rs]})

cut :: TxtEdit -> TxtEdit -- Cut highlighted item
cut (TxtEdit{l = ls, c = cp, r = rs, h = he}) = (TxtEdit{l = ls, c = cp, r = rs, h = he })

copy :: TxtEdit -> TxtEdit -- Copy highlighted item
copy (TxtEdit{l = ls, c = cp, r = rs, h = he}) = (TxtEdit{l = ls ++ he , c = cp, r = rs, h = he })

paste :: TxtEdit -> TxtEdit -- Paste highlighted item --Error occurs if new items are highlighted and then you paste, need a clipboard??
paste (TxtEdit{l = ls, c = cp, r = rs, h = he}) = (TxtEdit{l = ls ++ he , c = cp, r = rs, h = he })

backspace :: TxtEdit -> TxtEdit -- Delete character left of the cursor
backspace (TxtEdit{l = ls, c = cp, r = rs, h = he}) = (TxtEdit{l = (reverse(tail (reverse ls))), c = cp, r = rs, h = he})

delete :: TxtEdit -> TxtEdit -- Delete character right of the cursor
delete (TxtEdit{l = ls, c = cp, r = rs, h = he}) = (TxtEdit{l = ls , c = cp, r = (reverse(tail (reverse rs))), h = he})
{-
setFile:: TxtEdit -> String -> IO () -- Puts String into a file
setFile (TxtEdit{l = ls, c = cp, r = rs, h = he}) = ls ++ rs =

getFile:: FilePath -> IO String -- Pulls String from a file
getFile ("E:\\University Work") = [("Test")]
-}
{-
http://learnyouahaskell.com/
https://www.newthinktank.com/2015/08/learn-haskell-one-video/
http://cheatsheet.codeslower.com/CheatSheet.pdf
http://hackage.haskell.org/package/ansi-terminal-0.8.0.2/docs/System-Console-ANSI.html
http://hackage.haskell.org/package/base-4.11.1.0/docs/Prelude.html#v:interact
http://learnyouahaskell.com/syntax-in-functions
http://learnyouahaskell.com/making-our-own-types-and-typeclasses
https://wiki.haskell.org/Introduction_to_Haskell_IO/Actions
https://www.youtube.com/watch?v=_dUW7iWs0No
-}

--TxtEdit {leftSide = "hell", rightSide = "o",cursorLocation = 3,sentenceLength  = 5}
--let test1 = TxtEdit {leftSide = "hell", rightSide = "o",cursorLocation = 3,sentenceLength  = 5}
--let test1 = TxtEdit {l = "hell", r = "o",c = 3,s = 5}
--let test1 = TxtEdit {l = "hell", r = "o",c = 3,s = 5, h =""}
--let test2 = TxtEdit {"hell", "o", 3, 5}


--import System.Console.ANSI.Types

{-main = do
putStrLn("Test")
get <- getLine

putStrLn ("Hello World " ++ get)
-}

{-Login = do
putStrLn("Welcome to the text editor, please login")
```

```
putStrLn("Username:  ")
x <- readLn
if x == "John_0105" then

putStrLn("Password: ")
-}
{-main = do
  setSGR [SetColor Foreground Vivid Red]
  setSGR [SetColor Background Vivid Blue]
  putStrLn "Red-On-Blue"
  setSGR [Reset]  -- Reset to default colour scheme
  putStrLn "Default colors."
-}
-- | Char Char Char Char | Char String String String
--data Txt = TxtEdit Char Char Char Char | TxtEdit String String String String


{-addChar :: TxtEdit -> TxtEdit
addChar (l h a x) = ((l++['q']) h a x)-}


--test :: TxtEdit -> String
--test (TxtEdit a b c d ) = a ++ " "
{-cursorMoveLeft :: TxtEdit -> TxtEdit
cursorMoveLeft(TxtEdit  [] a b c) = TxtEdit [] a b c
cursorMoveLeft(TxtEdit  l  a b c) =
if head (reverse l) == '' && head(a++b) /= '' then TxtEdit  l a b c
else cursorMoveLeft(CurseLeft(TxtEdit  l a  b c))

-}
```