**Document Design**

**Design Overview:**

Here is the design section of the project, initially there were some changes to the original objectives set out in the analysis. In the design section we can also see the intricate object-oriented programming that was used for the solution of the problem to be solved. There are many different parts/sections to the solution that all contributed to making a complete solution the complete solution also has room for additional elements to be added in the future which allows the project to be infinitely expandable and adaptable to the new environment which it is needed in. There were three main things that overlooked the whole solution these were the: the data, the algorithms, the UI (User Interface). Below you can see the main programs hierarchy chart. This shows the main different stages of making the solution for this project. You can see that many of the sections break down into smaller sections, which they also break down into smaller sections, showing the complexity of this task. The data part which I said you could see represented by the Pull Data section, the algorithm section is shown in the algorithms and the analysis/display section. Lastly the UI section can be seen by the display part in the analysis/display section. However it is implemented throughout the code.

**Hierarchy Chart:**

Below you can see the hierarchy chart that shows you how the different parts of the project have been able to come together to make a final solution.

There is unit that connects to the main form called stock unit. This stores the classes of TData and TStock. Where its creation procedure is called in the main form, and assigned to an index in an array. The unit takes the unformatted data values and sorts them into a format that is actually viable for use in the program. This is then stored in an array so that this can be done for a multitude of stocks. The class TData is never called by the main program, that class is called by the class TStock when TStock is called by the main program.

Analysis/Display

Sorting

Suggested Portfolios

Suggested Buys/Sell

Stock Program

Main Program/Form

Stock Unit

Creating Objects Populating

Form Objects

Objects from Stock Unit

Pull Data

From downloaded .txt files

From internet on request (API)

Algorithms

Correlation

Covariance

Standard Deviation

Moving Day Average

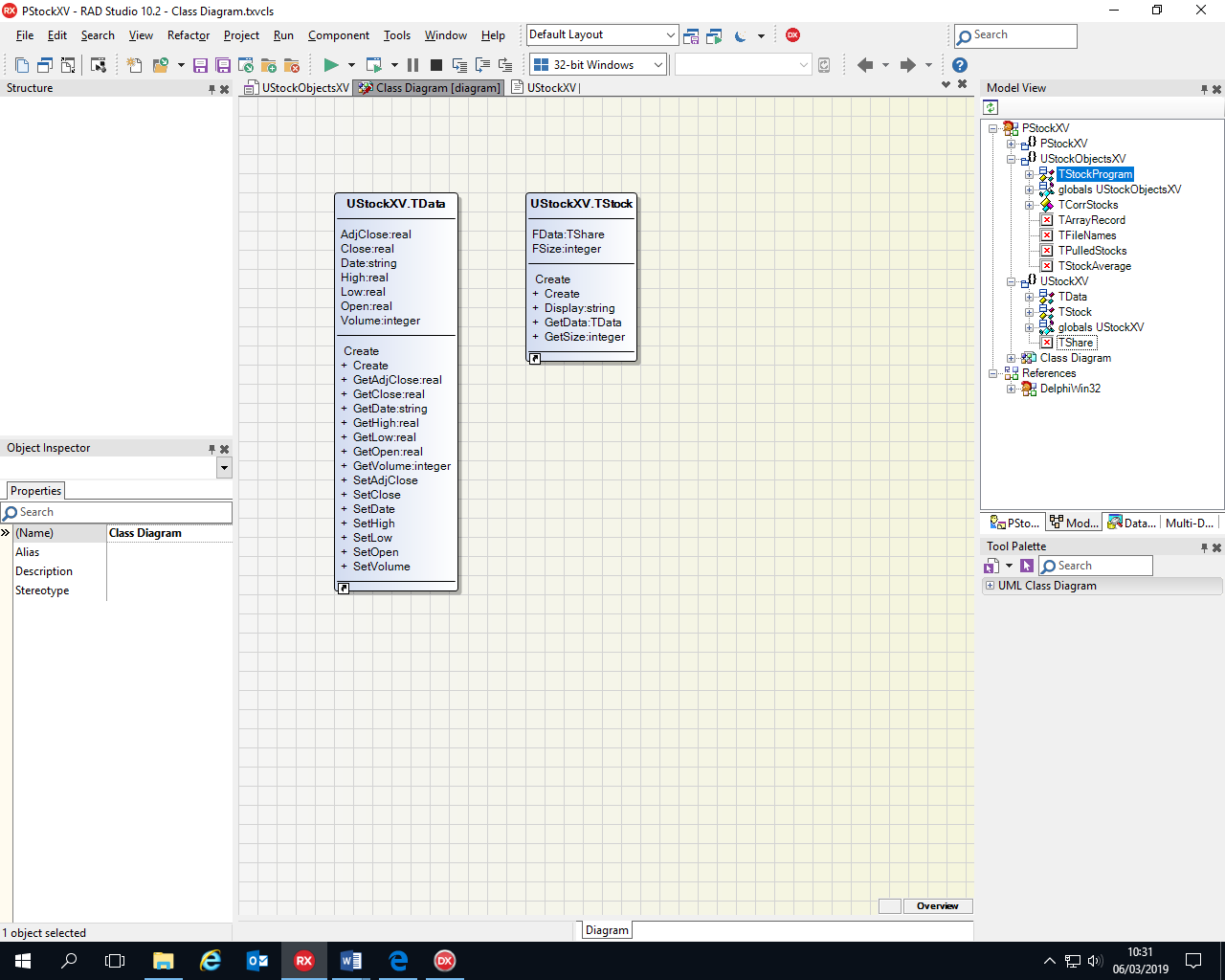
**Changes to Objectives:**

The client and I very quickly realised that some of the objectives were very ambitious in the time period and very challenging for an A-Level coursework project, additionally some objectives were just not important enough or too unrealistic, to be continued in the project. Some of these objectives were completely removed, whereas other were amended as they could still have significance to the project, furthermore more some extra objectives were added as the when the solution was being completed there was a realisation of a need of an extra part of the project.

The final objective: *“Actually execute purchases and sales on the stock market with real or fake money.”* Could only be fully done with a fully finished program so that was changed to being more of a test if possible than an objective that can be met. Because of this, this objective was formally taken out of the objectives however would be a nice test if the solution is fully completed.

**Summary of Key Variables/Types/Classes:**

|  |  |  |
| --- | --- | --- |
| Name of Variable/ Type | Data Type | General Overview |
| TData | Class | Where all the individual values for the data points are stored. The private variables that it contains are: date (string), open (real), high (real), low (real), close (real), AdjClose (real), and volume (integer). Stores a single piece of data’s values. Has an overloaded constructor in order create the object in the case it’s from a .txt file or from the API. |
| TShare | Array of TData | This contains an array of TData this full array gives all the pulled data points, of the time period that was in the data. |
| TStock | Class | This is the complete stock. Containing the private variables FData and FSize which you can see below. It is the direct object that is created by the main program, containing the links to all the necessary data points needed. |
| FData | TShare | This contains a dynamic array of TData this full array gives all the pulled data points, of the time period that was in the data. This now being apart of TStock allows TStock to inherit all the properties, and variables for TData |
| FSize | Integer | This variable contains the amount of data points there will need to be in the FData, set the length of FData. |
| TPulledStock | Array of TStock | Array of TStock from the Stock unit, which is connected to the code as you can see in the connections in the hierarchy chart. |
| MyPulledStocks | TPulledStock | This is where all the stock data is stored from the internet or the .txt file you can use this to access the open, close, high, low, volume etc. of the stock and additionally it’s sized. Dynamic array of TPulledStocks, connection to the object TStock from the main program to the unit. |
| TCorrStocks | Record | A record that contains the required information for the correlation of stocks. Contains the indexes of the stocks that is needed, the correlation coefficient between them, and the dates between where this correlation has been taken between. |
| TArrayRecord | Array of TCorrStocks | This is a dynamic array of the record TCorrStocks this is used to store all the possible combinations, of correlation between the stocks had. This is created from the corresponding type and thus it inherits its properties. |
| AllCorrStocks | TArrayRecord | This is the physical variable that is used in the algorithm that has been created from the previous data types, and thus inherits its properties. This array is also sorted. |
| MaxDataNum | Integer | This is contains the integer value for the maximum amount of stocks that can be contained. It is used to set the lengths of most of the dynamic arrays. The variable is initialised and thus defaulted to 5. |
| TFileNames | Array of string | This is a dynamic array that is used to store all the file names. The length is set by the variable MaxDataNum. |
| MyFileNames | TFileNames | This is the physical variable that is used by the main program. It has been created from the type TFileNames so inhierits all its properties. Such as being a dynamic array. Where the length is set by MaxDataNum. |
| StockIndexValues | Array of integer | This is a dynamic array of integers the length is set by the variable MaxDataNum. This array contains the position of all the stocks in MyPulledStocks. Mainly used to go to the positions of the stocks data. Ignoring positions where there is no stock values. |
| TStockAverage | Array of array of array of real | These are 3 dynamically created arrays that are used to store the moving day averages of the different stocks. Each stock has a high and low value. The first array is the corresponding stock the next array is the high and low averages, and the last array is filled with the values. The length of the arrays are set by the MaxDataNum, the next array length is set to 2 for high and low, and the last array is set to the length of the amount of averages. |
| MyStockAverage | TStockAverage | This is the physical variable that is used by the main program. It has been created from the type TStockAverage, thus it inherits all its properties. |

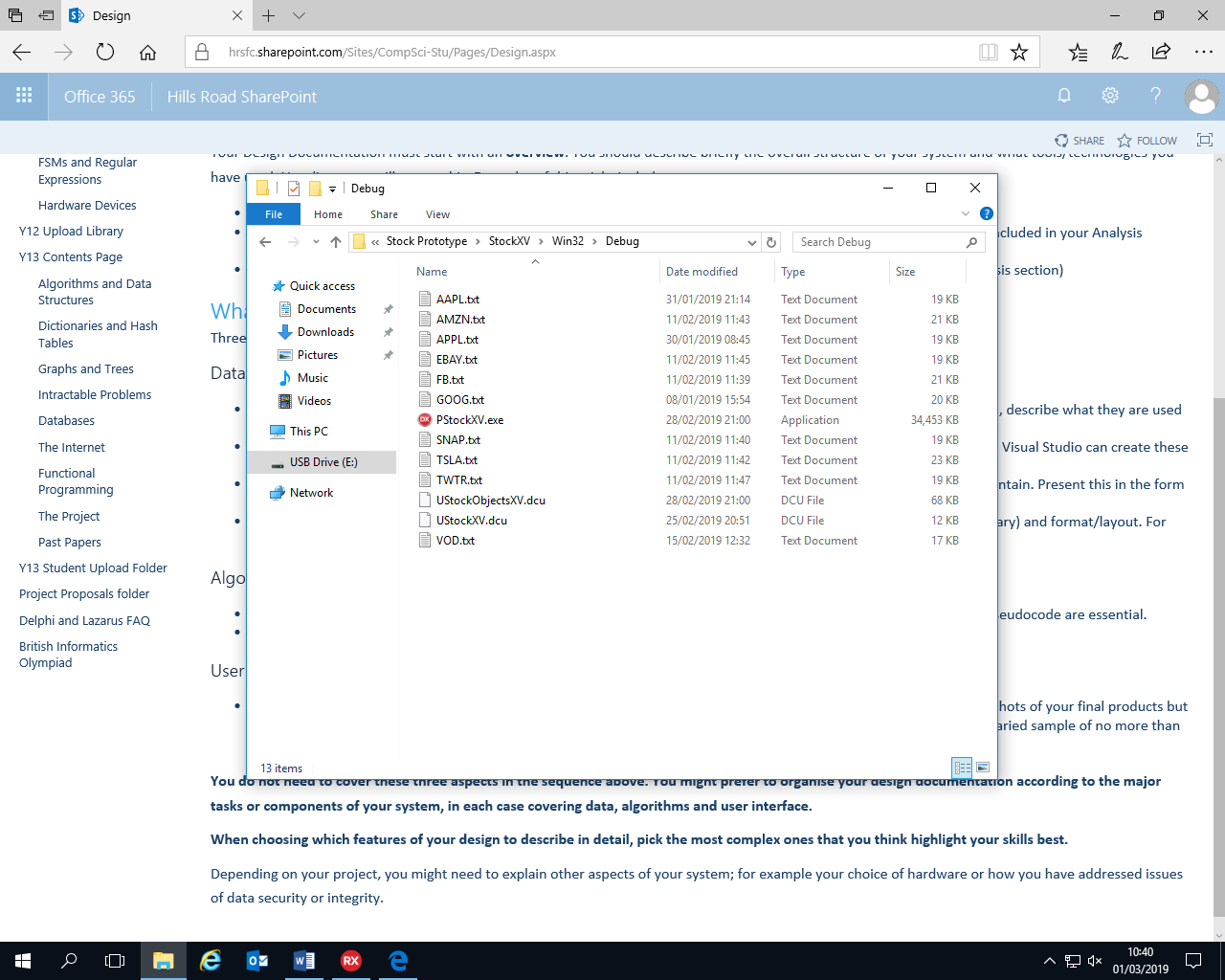


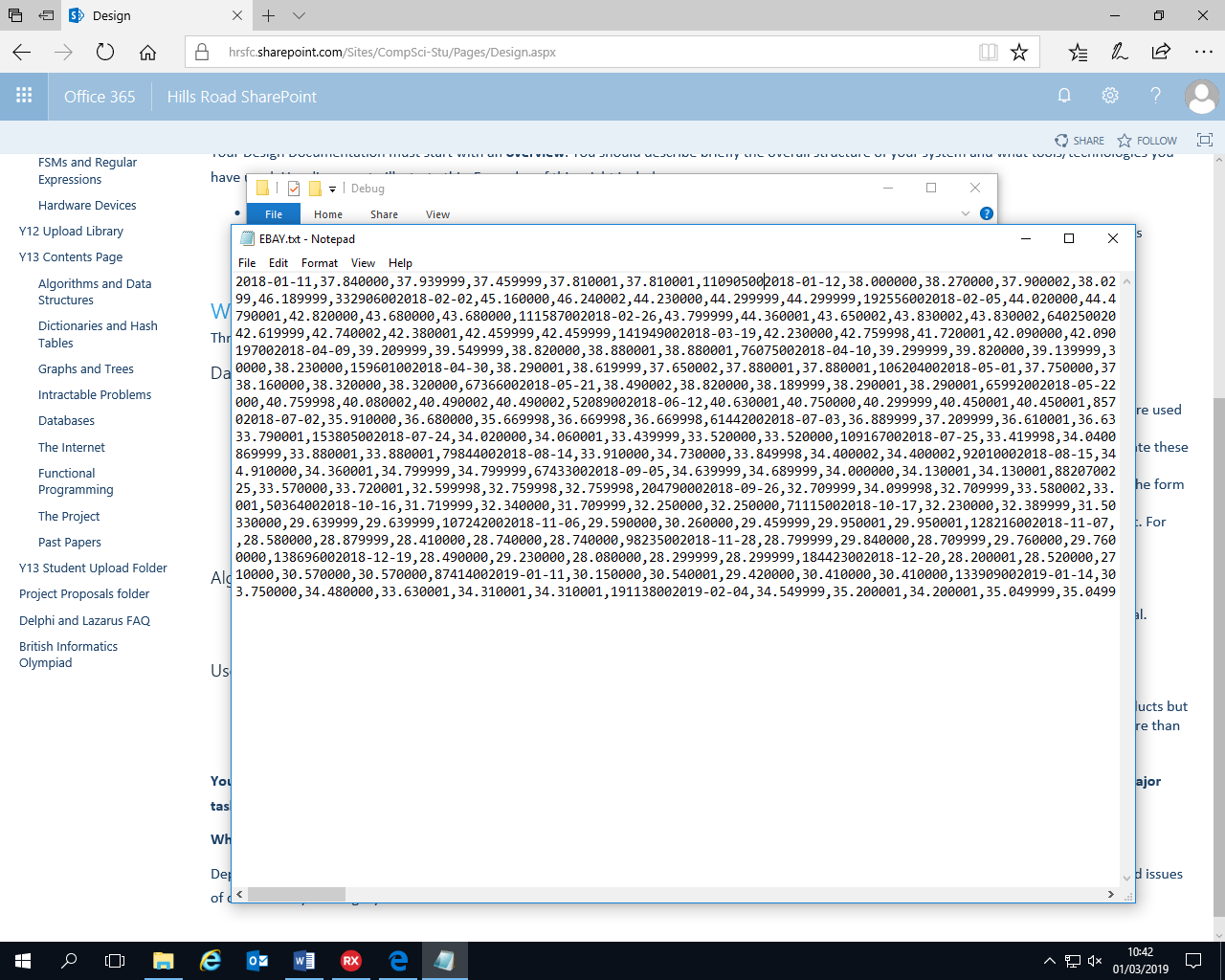
This is the class diagram of the classes in the program. As you can see there are 2 classes which are TData and TStock.

TStock contains the variable FData and Fsize. FSize contains the amount of data points that are contained in the stock that has been pulled. While FData is an array of all the data points and all the corresponding values in the data points. The procedures that this class contains is its constructor create this has 2 different version to handle if it is creating a stock from data from the API or data from a .txt file. It then allows the user to call the function getSize which returns the amount of data points there are or, getData which allows the user to choose which data value that they want to return.

TData is the other class and it contains the varaibles: AdjClose, Close, Date, High, low, open, and volume. These are the different data values that a single data point contains. It has 2 constructors to create the data point from the API or the .txt file depending on the parameters inputted. It then can get the individual values and return them to the function that called them, e.g. the open of a specific data point. The set values are empty procedures that were thought to be useful however now all of their contents are commented out, as the user shouldn’t be editing the data pulled in the stock.

**File Organisation**





This shows the file organisation of all the .txt files that are needed if the user has decided to download the .txt file and use it instead of getting it from the API. The .txt file needs to be saved in the directory where the program file is saved then /Win32/Debug. This is directory that is needed for the files to be accessed. Also depending on the source of the file the format will be in a .csv, this needs to be changed into a .txt. The source of my files were found by the following URL: “https://uk.finance.yahoo.com/quote/GOOG/history?p=GOOG”. Additionally the file automatically went to my Downloads file, so the destination had to be changed to the correct, directory. By changing the file type of the file. It is still in the csv format however now just as a readable .txt file. The file name is downloaded as the Ticker symbol + .csv, you can change it to a custom name but it is the name you will have reference when calling the .txt file in the program. This directory is also where the program executable file is stored, and additionally the DCU file of the unit and the main code. Therefore the only thing the user needs to do is download the file change it to a .txt and then move it into the correct directory.

The .txt file is in a csv format so all the values are separated by commas, and due to it being meant to be in a .txt file it has all the corresponding next line code imbedded in the .txt file. So this is used to by the program to differentiate between the data points. The file values start go in this order: date, open, high, low, close, adjclose, and lastly volume. There are 7 values which the program also takes into account and uses to correctly organise the data. The data in the file is all a string, however in the program it correctly puts it into its correct data types. The program iterates through each line, which is one data point, it then adds character by character to a string variable until it reaches a comma or the end of line. In these moments the string variable is then saved to the corresponding variable, and it changes it into the correct data type for the variable this being: string, real, integer. The chunk is then set to an empty string so it can start adding the characters again the programs also increments the variable that tells the program which data point it corresponds to. The data in this file isn’t meant to be edited which is why the format of the file doesn’t matter as this type of file would be prone to human error when editing.

**Algorithms**



This algorithm is used to set the data values for a single data point. Setting the open, high, low, close, adjclose, volume. It takes in the line(string) of the all the values in the line of the .txt files and correctly assigns them to their destination variables.

Steps:

1. Initialises the order of the variables being sorted. So it starts at the right place.
2. Iterates between the first character and the last.
3. Add characters up until the program reaches a comma or the end of line
   1. If a comma is found then check which variable receives this value.
   2. Then set it to the variable doing any necessary data type changes.
   3. The change the indexer to the next variable to the receive the value.
   4. Reset the contents of the value to an empty string ready for the next set of character.

***Pseudocode:***

FUNCTION

TDataConstructor1(Line, count)

#Initialise chunk to an empty string.

STRING Chunk🡸 ‘’

#Initialise the variable that dictates the which variable chunk goes to.

INTEGER i🡸 0

#Iterate from the beginning of the string to the end

FOR INTEGER s 🡸 1 TO count

#When the code reaches a comma or the end of the line

IF (Line[s]= ‘,’ ) or (s=count) THEN

#Checks i to decide which variable gets the congregated characters.

IF i=0 THEN

STRING Date🡨 Chunk

ENDIF

IF i=1

REAL Open🡸 STRING\_TO\_REAL(Chunk)

ENDIF

IF i=2

REAL High🡸STRING\_TO\_REAL(Chunk)

ENDIF

IF i=3

REAL Low🡸STRING\_TO\_FLOAT(Chunk)

ENDIF

IF i=4

REAL Close🡸STRING\_TO\_FLOAT(Chunk)

ENDIF

IF i=5

REAL AdjClose🡸0

ENDIF

IF i=6

INTEGER Volume🡸STRING\_TO\_INT(Chunk)

ENDIF

#Increments i to get to the next variable, and resets chunk to an empty string

INTEGER i🡸i+1

STRING Chunk🡸 ‘’

ELSE

#if the character isn’t a comma or at the end of the line, add the character to chunk.

STIRNG Chunk🡸 Chunk+Line[s]

ENDIF

ENDFOR

ENDFUNCTION



This algorithm is used to set all the data values for all the data points, setting the open, high, low, close, adjclose, volume, and adding them to an array the length of the inputted size of the .txt file which is the number of lines. It takes in the whole file name with the extension (.txt), to find it in the debug folder. It also takes in the how many lines, which is how many data points there are, in the file.

Steps:

1. Sets the variable FSize to the passed in variable arrayLength
2. Sets the length of the array(FData) to the correct variable (FSize).
3. Checks if the file exists
   1. If it does assign the file so it can be used.
   2. Open the file to allow it to be read.
   3. Initialises the variable that dictates the index of a single data point to 0.
   4. Loops till it gets to the end of the file
      1. Reads a line of the file.
      2. Checks the length of the line is greater than 0.
         1. If it is then set count to the length of the line.
         2. Add the data points to the array, in the right format by calling the subroutine TData.Create(line, count).
   5. Close the file to stop corruption.

***Pseudocode:***

FUNCTION

TStock.Create(fileName, arrayLength)

#Set the length of the variable FSize to the passed in parameter arrayLength.

INTEGER FSize🡸 arrayLength

#Sets the length of the variable FData to FSize, so all data points occupy an index

LEN(FData)🡸FSize

#Check that the file exists

IF FILEEXISTS(fileName)= True THEN

#Open the file for reading. The file is the name passed in as a parameter

myFile🡸 openRead(fileName)

#Initialise the index of where the first set of datapoints will a go

INTEGER i🡸0

#Read every line of the file one at a time

WHILE NOT myFile.endOfFile()

#Get a line out of the file

STRING Line🡸myFile.readLine()

#Check if the line is empty, do the next steps if it isn’t empty

IF LEN(Line)>0 THEN

#Set the count variable to the length of line

INTEGER Count🡸LEN(Line)

#Attach the created data point to the index which is i

FData[i]🡸TData.Create(Line, count)

#Increment the integer

INTEGER i🡸i+1

ENDIF

ENDWHILE

myFile.close()

ENDIF

ENDFUNCTION



This algorithm is used to set all the data values for all the data points, setting the open, high, low, close, adjclose, volume, and adding them to an array the length of the inputted size of the .txt file which is the number of lines. It takes in all the data points which is an array of string. 1 index of the string contains all the necessary values for a single data point. It also measure the length of the array and sets the size of the stock to the length of the array.

Steps:

1. Sets the variable Fsize, which dictates the size of the stock, to the length of the array.
2. Sets the length of the array of all the data points FData, to the correct size FSize
3. The program iterates through all the indexes in the dynamically sized array FData.
   1. In each iteration it adds the data points to the array, in the right format by calling the subroutine TData.Create(LiveData[i]).

***Pseudocode:***

**Function**

TStock.Create(LiveData)

#Set the length of the variable FSize to the length of the array.

**integer** FSize🡸 LEN(LiveData)

#Sets the length of the array of data points to the FSize.

LEN(FData)🡸FSize

#Iterate through each and every index of FData.

**For** **integer** i 🡸 0 **to** FSize-1

#Add the data points in a format to the specific index in FData

FData[i]🡸 TData.Create(LiveData[i])

**EndFor**

**EndFunction**



This algorithm is used to set the length of all the array lengths in the program. It intakes the number in the edit box MaxDataEdit.Text. It also error checks the number in the edit box and handles if an error is encountered. If a number is given that is less than 5 the lengths of all the array is set to 5.

Steps:

1. Try do the following.
   1. If the number in the edit box is greater than 5 do the following.
      1. Set the variable MaxDataNum to the number in the edit box.
      2. Set the length of array MyFileNames to the variable just set.
      3. Set the length of array MyPulledStocks the variable just MaxDataNum.
      4. Set the length of array MyStockAverage the variable just MaxDataNum.
      5. Set the length of array StockIndexValues the variable just MaxDataNum.
      6. Change the text in the edit box to “Yes”.
   2. Change the text in the edit box to 5.
   3. Reset all data in MyPulledStocks.
2. If error encountered, show an error message.

***Pseudocode:***

**Procedure**

TStockProgram.MaxDataBttnClick(Sender)

#check if it can do the input of the edit box

**Try**

**#**take the input of the text in the edit box

**string** MaxDataEdit.Text 🡸 Input

**#**If the input is greater than 5

**If** **string\_to\_integer**(MaxDataEdit.Text)>=5 **then**

**#**Set the variable to the input

**integer** MaxDataNum🡸**string\_to\_integer(**MaxDataEdit.Text**)**

**#**Set all the lengths of the arrays

Len(MyFileNames)🡸 MaxDataNum

Len(MyPulledStocks)🡸MaxDataNum

Len(MyStockAverage)🡸 MaxDataNum

Len(MyStockAverage)🡸 MaxDataNum

**ELSE**

#If it isn’t greater than 5 output 5 in the edit box

**string** MaxDataEdit.Text🡸’5’

**EndIf**

#Reset all values is the array below to nil. Deletes all contents

**For integer** i **🡸** 0 **to** Len(MyPulledStocks)-1

MyPulledStocks🡸 nil

**EndFor**

#If an error has occurred during a method then

**OnError**

#tell the user what they need to enter

Output(‘Enter an integer’)

**EndTry**

**EndProcedure**



This algorithm is used to pull the data from the internet and pull it into the array MyPulledStocks, in the correct index. It also pulls the data from .txt file, and puts it in the array with the correct index. It takes the name of the stock that they want to pull from an edit box (StockNameEdit), and it determines the index of the stock with an edit box (QuantPullEdit). It calls the create function of TStock from the other unit in my program, which correctly formats the data. It also connects to the API by forming the client, request response connections.

Steps:

1. Initialise variables required, and clear display for showing stocks.
2. Try to get an integer value of the stock index if not a message is given to the user to change that value.
3. Check if the file exists if so, it does the following.
   1. Assign the file to a variable so it can be used.
   2. Open the file for reading.
   3. Finds the length of the file, and passes it into a variable. (How many data points).
   4. Closes the file to stop corruption
   5. Set the correct index of the array to the created TStock which contains all the data points.
   6. Program notes that it was successful in a variable.
4. If the file doesn’t exists try to check for it online with the API.
   1. Get the time period they want to take the code over. Default is 6m (6 months).
   2. Make the request URL.
   3. Try and execute the request if it can’t be executed then output a message saying it can’t be found. Program notes that it was unsuccessful.
   4. If successful then it does the following.
      1. Sets the field names, and initialise variable used for finding amount of data points.
      2. Go to the first data value.
      3. Iterate through all the data finding out the amount of data points.
      4. Go back to the first data value.
      5. Sets the length of a temporary array of record to the amount of data points found.
      6. Reset i to 0, for the iteration through the data values again, and start iterating.
      7. Add each data point the correct field in the record, and in the right index of the array.
      8. Set length of a temporary array to the length of the previous temporary array.
      9. Temporary array takes a string value of all the data values for a single data point, separated by commas.
      10. Create the stock at the index of MyPulledStocks.
      11. Set success to true
5. If successful then it does the following.
   1. Adds the file name to the MyFileNames array.
   2. Increment the counter.
   3. If the counter is greater than the max no. of stocks wraparound to 0.
   4. Set the edit box to the new counter.
   5. Change the name of the file name edit box StockNameEdit to success

***Pseudocode:***

**Procedure**

TStockProgram.PullDatBttnClick(Sender)

#Initialise variables and set up records

**record** TRecord fields: **string** date, **real** open, high, low, close, **integer** volume.

a🡸 array of TRecord

**boolean** apiPull🡸True

**boolean** success 🡸 False

#Clear the memo and enter a heading

Outputs to memo 🡸(Empty **string**)

Output to Memo(‘StocksPulled’)

#take in the input of the file name

**String** temp🡸 Input StockNameEdit.Text

#try to get the index to put of where to put the data

**Try**

**Integer** counter🡸 **string\_to\_integer(**Input QuantPullEdit.Text**)**

#if there is an error

**OnError**

Output(‘Error need integer’)

**EndTry**

#Check if the file exists

**If FILEEXISTS(**Input StockNameEdit.Text) **then**

dataFile1🡸 openRead(Input StockNameEdit.Text)

#Get the length of the file how many data points

**integer** ListArrayLength🡸 **function** FindLength()

#Store the formatted data points in the correct index of the array

MyPulledStocks[counter]🡸 **function**

TStock.Create(StockNameEdit.Text, ListArrayLength)

#Set success to be true

**Boolean** success🡸True

**ELSE**

#Default string value of time frame of API stock

**String** temp2🡸 ‘6m’

#Set the time frame of the stock you want to pull

**In the case of** RadioGroup.ItemClicked = x **then**

0: **string** temp2🡸 ‘5d’

1: **string** temp2🡸 ‘1m’

2: **string** temp2🡸 ‘3m’

3: **string** temp2🡸 ‘6m’

4: **string** temp2🡸 ‘1y’

5: **string** temp2🡸 ‘2y’

6: **string** temp2🡸 ‘5y’

**EndCase**

**#**Make extension to URL so it can be pulled

Request.URLExtenstion🡸 **string** temp+ ‘chart’ + temp2

#Try and execute the request on the given URL

**Try**

**Execute.**request

**OnError**

**#**If file not found

OUTPUT(File not found)

**Boolean** ApiPull🡸False

**Boolean** success🡸 False

**EndTry**

#If the flie was found

**If** ApiPull=True **then**

#Set all the fields variable to the fields in the data source and initialise

Fields.All🡸Datasource[(fieldnames)]

**Integer** i🡸0

**#Find amount of data points in the file**

**While NOT** DataSource.eof

**Integer** i🡸i+1

Datasource.Next

**EndWhile**

LEN(A)🡸 i

**Integer** i🡸 0

#Go to the first point in the data source

Datasource.First

#Add the bits of data to the records in the correct index of the array

**While NOT** DataSource.eof

**String** a[i].Date🡸field1.Value

**real** a[i].Open🡸field2.Value

**real** a[i].High🡸field3.Value

**real** a[i].Low🡸field4.Value

**real** a[i].Close🡸field5.Value

**integer** a[i].Volume🡸field6.Value

**#**Go to the next line in the Data source

Datasource.Next

**EndWhile**

Len(b)🡸Len(A)

#Add all the values in one data point to a comma separated line

**For integer** I 🡸 0 **to** Len(b)-1

**String** Temp3🡸 a[i].allRecords

B[i] 🡸 temp3

**EndFor**

#Created formatted stock in the correct index and set successful.

MyPulledStocks[counter]🡸 TStock.Create(b)

**Boolean** success🡸 True

**EndIf**

**EndIf**

#If successful then add the name of the stock to the array that stores all the names

**If** success=True **then**

MyFileNames[counter]🡸temp

**Integer** i🡸i+1

#If counter has gone over the max amount of data slots available reset to 0

**If** counter>(MaxDataNum-1) **then**

**Integer** counter 🡸0

**EndIf**

#Show new value of stock index and show user that it was a success

OUTPUT(**integer\_to\_string(**counter) **to** QuantPullEdit.Text)

OUTPUT (‘Success’ **to** StockNameEdit.Text)

**EndIf**

#Show all the pulled stocks to the user

**For integer** i🡸0 **to** LEN(MyFileNames)-1

OUTPUT(**integer\_to\_string(**i**)** + ‘: ‘+ MyFileNames[i])

**EndFor**

**EndProcedure**



This algorithm is used to make the list of average that is used to plot an average line as well. Additionally this algorithm is also used in the simple moving day average function. It gives the values of the stock’s average depending on the day average period you want to put it over. It accommodates for putting a high and a low value which is the main part of the algorithm which is used for the simple moving day average.

Steps:

1. Initialises all the variable assigning the period day value
2. Set the length of the 3rd tiered array that will be holding the data to the amount of data points that there will be for the average.
3. Iterates through the values of the data value.
   1. Initialises total.
      1. Adds all the values up in the period for a total.
   2. Divides the total by the period to get the average over the period.
4. Stores the average in the correct index of the averages array.
5. Increment the lower and the only increment the higher(if indexing to a point still in the array)





These are 2 identical algorithm that use the same methods however stores their data in different corresponding points, plots their data using different corresponding series on the chart, and have different colours. These methods first of all delete any initial lines, before adding the new line. They give the series an appropriately named legend, so that the user knows what the lines is that is plotted. It also sets the length of the 2nd tier array in the averages array to 2 to accommodate for a higher and lower average to be populated. It then calls the function that makes the average list, with the correct time period. It then plots that data. If an error occurs due to the day period input of the user the error is handled and a message is shown to the user, alerting them of this.

Steps:

1. Clear the line that is already on the graph, if there is.
2. Get the day period that the user wants to make the average over.
3. Get the stock that they want to do this with.
4. Sets the length of the 2nd tiered array in the averages array to 2.
5. Calls the function to make the averages list.
6. Then plots the averages list
7. If no errors have occurred, if so the following is done
   1. They are either told to enter a value greater than 0 or to enter an integer value dependent on what they have entered.



The algorithm shown in the picture above is used to display all the data values of the chosen stock. The stock desired is found out by the program by seeing the number that they have entered in the edit box and trying to find the data for the index of the data it then adds all the data points line by line, to the memo.

Steps:

1. Gets the index value of the stock and converts it from a string to an integer.
2. Iterates through all the data points displaying each data point and all the values.
3. Handles the error if the input is not valid.



The algorithm above display a single line and all its data values for a single point. It takes the input of a line number and of which stock that you want the data displayed. These values are entered into different edit boxes and are validated for correct input when a button is clicked. It adds the values for these to the memo. Which shows the name of the stock and all of its other relevant values.

Steps:

1. Gets the index value of the stock that is desired and converts it into string.
2. Displays the data of the line using the value of the line as a variable passed into the displaying of the data.
3. Handles error if the input isn’t valid.



This algorithm is the algorithm that gets all the data for the correlation algorithm to take place it also displays the value of the correlation coefficient that is received. This procedure takes in which stocks it will test correlation coefficient with. The date that it wants to test between if there is one. And will handle all the errors that may come with trying to carry out these methods.

Steps:

1. Gets the user entered indexes and dates from the edit boxes.
2. Format the date, in the necessary file.
3. The program tries to find the dates in both stocks.
4. If no date is entered the program tries to find the maximum date period between the 2 stocks.
5. Checks if a date is entered that there is a date in both cases, if not output an error that no common date is found the files.
6. If there is dates found in all the stocks then call the correlation function passing in the parameters of both indexes of the stock and the date lines of the dates.
7. If the correlation coefficient is between -1 and 1, then the program does the following.
   1. Gets the start date and the revised end date and assign them to variables.
   2. Display the 2 stocks and the correlation coefficient and the dates that it was taken over.
8. Handles errors that come up due to the users input by showing a message.



This is the algorithm that actually physically calculates the correlation coefficient between the 2 stocks it calls a multitude of other functions and procedures that are needed to calculate the correlation coefficient, such as the covariance, the means of both stocks and their standard deviations. It also find the maximum possible dates if no specific date is passed in, which would have been by the user.

Steps:

1. Initialise all variable at the beginning.
2. Checks which dateline variable value is lower then sets variables values based on that.
3. If both starting dateLines passed in are 0 which most likely means the user didn’t enter a date.
   1. Then the lowest common date is found between the 2 stocks by calling the function CorrleationDateLimit
4. If both ending dateLines in are the maximum date line it most likely means that the user didn’t enter a date.
   1. Then the highest common date is found with the same function as before but by passing in different parameters, the function is CorrelationDateLimit.
5. If 2 common dates were found then the program does the following.
   1. Sets the lowest and highest value for the different stocks.
   2. Calculates the line difference between common dates in the stocks as well.
   3. Sets the dateline variables.
   4. Then the program calculates the mean of both stocks between the dates and stores each value from a function.
   5. The program then calculates the standard deviation of both stocks between the dates and stores each value, from a function.
   6. It can then use these values to work out the covariance of the 2 stocks, between the dates and store the value, from a function.
   7. It can then use these worked out values to calculate the correlation coefficient of the 2 stocks.
6. If no common date the result is 2 because it clearly isn’t a correct stock correlation number as all correlation coefficients are between -1 and 1.

***Pseudocode:***

**Function**

**#**Function to get the correlation coefficient between stocks

Correlation(**integer** index1, index2, dateLine1, dateLine2, dateLine3, dateLine4)

#Set up record

**Record** CorrStock1.Fields🡸 **integer** index, low, high, **real** mean, sigma

**Record** CorrStock2.Fields🡸 **integer** index, low, high, **real** mean, sigma

#Initialise variables to default values, chosen specifically

**Integer** higherIndex 🡸 -1

**Integer** lowerIndex 🡸 -1

**Integer** CorrStock1.index🡸 index1

**Integer** CorrStock1.low🡸 dateLine1

**Integer** CorrStock1.high🡸 dateLine3

**Integer** CorrStock2.index🡸 index2

**Integer** CorrStock2.low🡸 dateLine2

**Integer** CorrStock2.high🡸 dateLine4

#Setting which variable has a lower line value and thus higher value index

**If** dateLine1<dateLine2 **then**

**Integer** lowerlimit🡸 dateLine1

**Integer** higher 🡸 dateLine3

**Else**

**Integer** lowerlimit🡸dateLine2

**Integer** higher🡸 dateLine4

**Endif**

**#**Finds minimum common start date and maximum common end date

**If** (**integer** dateLine1=0)AND (**integer** dateLine2=0) **then**

**Integer** Lowerlimit🡸 Function CorrelationDateLimit(index1, index2, 0, 0,

False, lowerIndex)

**Endif**

**If** (**integer** dateLine3=LEN(MyPulledStocks[index1])-1)AND (**integer** dateLine4=

LEN(MyPulledStocks[index2])-1)) **then**

**Integer** higher🡸 Function CorrelationDateLimit(index1, index2,

LEN(MyPulledStocks[index1])-1, LEN(MyPulledStocks[index2])-1,True, higherIndex)

**Endif**

#If there is a stock correlation date overlap

**If** (higher>=0) and (lowerlimit>=0) **then**

**#**Change the low and high value for the respective stocks

**if** higherIndex=index1 **then**

**integer** CorrStock1.high🡸 higher

**else** higherIndex=index2 **then**

**integer** CorrStock2.high🡸 higher

**Endif**

**if** lowerIndex=index1 **then**

**integer** CorrStock1.low🡸lowerlimit

**else** lowerIndex=index2 **then**

**integer** CorrStock2.low🡸 lowerlimit

**Endif**

**#**Change the dateline values for the external caller and see difference in lines

**Integer** lineDiffer🡸 CorrStock2.low-CorrStock1.low

**Integer** dateLine1🡸 CorrStock1.low

**Integer** dateLine2🡸 CorrStock2.low

**integer** dateLine3🡸 CorrStock1.high

**integer** dateLine4🡸 CorrStock2.high

#Calculate means and standard deviations of both stocks

**real** CorrStock1.mean🡸 **function** Mean(CorrStock1.index, CorrStock1.low,

CorrStock1.high)

**Real** CorrStock2.mean🡸 **function** Mean(CorrStock2.index, CorrStock2.low,

CorrStock2.high)

**Real** CorrStock1.sigma🡸 StandardDeviation(CorrStock1.index,

CorrStock1.mean, CorrStock1.low, CorrStock1.high)

**Real** CorrStock2.sigma🡸 StandardDeviation(CorrStock2.index,

CorrStock2.mean, CorrStock2.low, CorrStock2.high)

#Calculate the covariance between the stocks

**Real** cov🡸 CoVariance(CorrStock1.index, CorrStock2.index,

CorrStock1.mean, CorrStock2.mean, lineDiffer, CorrStock1.low,

CorrStock1.high)

#Calculate the correlation coefficient and return it to the calling function

**Real** result🡸 cov/(CorrStock1.sigma\*CorrStock2.sigma)

**Else**

**#**If the stocks didn’t overlap in dates

**Real** result🡸 2

**EndIf**

**EndFunction**



This algorithm is used to find the mean of a stock between the time period. The upper bound of the time period is the variable higher which is passed into the function as a parameter. The lower bound is the value dateline which is passed into the function as a parameter. The stock index of the stock which the user wants the mean is also passed in so the mean is taken over the right stock. The value returned by the function is the mean.

Steps:

1. Initialise the total to 0, to the total to be made, accurately.
2. Iterate between the lower bound and the higher bound, doing the following.
   1. Adding each value to the total, getting the grand total.
3. Return the value of the grand total divided by the amount of data points the total was made over.



The algorithm standard deviation is used to find the standard deviation of the desired stock between the desired time frame, high and low values. These values are passed into the function, as parameters that the function expects when it is called. It also requires the mean of the stock index. This is passed in and calculated from a previous function.

Steps:

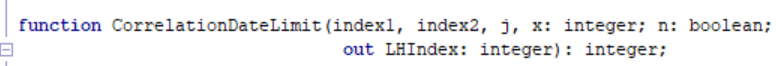
1. Initialise the variables needed.
2. Iterate between the time frame, high and low values, doing the following.
   1. Calculates the value of the data point minus the mean.
   2. Square this value and add it to the sigma total.
3. Return the value of standard deviation by square rooting the sigma total, which was variance.



This function works out the covariance between 2 stocks, by using both means of the stocks, the difference between the line values in the stock line indexing, and the lower bound and higher bound (which gives the time period). All the values gives the covariance and the variable mentioned are passed in by parameters in the function.

Steps:

1. Initialise the variables.
2. Iterate between the time period of which the covariance is being calculated, and do the following.
   1. Set the line value which gives the same date in the 2nd stock.
   2. Totals up data of stock 1 minus its mean, multiplied by the data in stock 2 minus its mean
3. Returns the value of the totalled total in 2.b. divided by the time period the covariance was taken over, which is the covariance.



This algorithm handles the accuracy error that would occur if 2 stocks are tried to be correlated without the indexes being perfectly aligned. Depending on the Boolean value n which is a parameter it finds the minimum common starting date, or the maximum common ending date. A variable that is passed is the value of the minimum index of each stock date, the stock indexes and the value that says which stock starting or ending index needs to be changed. This is variable has an out by it so it changes the variable that was passed in its placed, in the location where it was called.

Steps:

1. Get the minimum or the maximum date of the stocks.
2. Separate those date into year, month and day integer variables.
3. If the n is false then it works out the lowest starting date, so the programming does the following.
   1. Finds which date is greater.
   2. Then finds the lower date in the other stock, by the function DateIndexFind, it returns this value to the function that called it.
      1. The function then sets the variable that tell the function which stocks starting reference index needs to be changed, to the stock that contained the lower date.
   3. If they are the same it returns the value maximum index, which is the last index. it doesn’t change the variable that tell the function which stocks starting reference index needs to be changed.
4. If the n is true then it works out the greatest ending date, so the programming does the following.
   1. Finds which date is greater.
   2. Then finds the greater date in the other stock, by the function DateIndexFind, it returns this value to the function that called it.
      1. The function then sets the variable that tell the function which stocks starting reference index needs to be changed, to the stock that contained the lower date.
   3. If they are the same it returns the value 0, which is the starting index, in both stocks and it doesn’t change the variable that tell the function which stocks starting reference index needs to be changed.



This algorithm finds the index of storage in the array of the date passed into the function, in a particular stock. It iterates through the stock and returns the value of the index of the stock if the date is contained in that stock. If the date isn’t stored in the array then function returns the value -10 as the index This will be handled by the function that called this function as -10 isn’t a valid index.

Steps:

1. Initialises the variable of the index to -10.
2. Iterates through the stock trying to find the date.
3. If the date is found change the index variable of -10 to the correct value and return the variable.



This function checks the amount of stocks that are actually populated in the array that stores the stock as not all positioned will be filled. Additionally the algorithm also finds the index of the stock and adds them to a global array which is used in the main program. This is because the user could have a stock in 0,3,4. Which allows the program to call stocks from incrementing.

Steps:

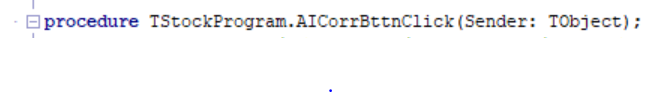
1. Initialise variables to 0.
2. Iterate through all the indexes of the MyPulledStocks which contain all stock data.
3. If stock data is found then the variable that tracks the amount of populated stocks increase by 1.
4. And add the position of that stock to the global array that stores all the positions of all the actually populated stock data.



This algorithm is used to find the amount of different combinations there are between the number of stocks that the user has. One of the parameters of the function is that it takes in the number of stock that the user actually has. It works out the different number of combinations excluding repeats and a stock combining with itself being counted. This value is returned to the function that called it.

Steps:

1. Initialises the counter variable, so its starts from 0.
2. Iterate through the stocks indexes, doing the following.
   1. Set the second stock value to be the current 1st iteration index +1.
   2. Iterate through number of possible second stock values, which was dictated by 2a.
   3. Add 1 to the counter for every iteration.
3. Return this value to the function that called it.



This algorithm works out all the correlation coefficient for all the stocks that have been pulled by the user. And puts them in an ordered array from highest positive to highest negative. It stores the values in an array of a record. The record values being the index of the 2 stocks, the correlation coefficient between them and the dates it occurred over.

Steps:

1. Initialise all necessary variables.
2. Call the function CheckPopStock to find out the amount of populated stocks assign value to a variable.
3. Get the different amount of combinations of stock by calling PascalsLength function, and assigning it to a variable.
4. Set lengths all necessary dynamic arrays.
5. Reset all record values in the all array indexes to null values.
6. Via iteration get all the different types of stock combinations.
   1. For each different combination call the correlation function, with a minimum and maximum date to be found.
   2. If the correlation coefficient is valid, then the program assigns all the values of the record to the correct value, in the correct index of the array.
7. Reduce the length of the array to conserve memory of all the stocks that didn’t overlap in dates with each other.
8. Merge the array of records by calling the function PerformMerge.

***Pseudocode:***

**Function**

AICorrBttnClick()

#Initialise variables and find out the amount of spaces needed in the array

**Integer** n🡸 0

**Integer** InvalidCount🡸 0

**Integer** numStocks🡸 **function** CheckPopStock()

Len(StockIndexValues) 🡸 numStocks

**Integer** recordLength🡸 PascalsLength(numStocks)

Len(AllCorrStocks)🡸 recordLength

AllCorrStocks.AllValues🡸(Reset)

#Iterate through all the different stock combinations getting all their correlations and putting them in the list

**For** **integer** i 🡸 0 **to** numStocks-1

**Integer** Paslow🡸 i+1

**For** j 🡸 paslow **to** numStocks-1

**String** StockCorr1Edit.Text🡸**integer\_to\_string**(StockIndexValues[i])

**String** StockCorr2Edit.Text🡸**integer\_to\_string**(StockIndexValues[j])

**Integer** dateLine1🡸0

**Integer** dateLine2🡸0

**Integer** dateLine3🡸 Len(MyPulledStocks[StockIndexValues[i]])-1

**integer**dateLine4🡸 Len(MyPulledStocks[StockIndexValues[j]])-1

**real** corr🡸 Correlation (StockIndexValues[i], StockIndexValues[j],

dateLine1,dateLine2, dateLine3, dateLine4)

**if** (corr<=1) AND (corr>=-1) **then**

**with** AllCorrStocks[n] **do**

**integer** Stock1🡸 StockIndexValues[i]

**integer** Stock2🡸 StockIndexValues[j]

**real** CorrValue🡸corr

**string** startDate🡸

MyPulledStocks[StockIndexValues[i]].GetData(dateLine1).GetDate

**String** endDate🡸

MyPulledStocks[StockIndexValues[i]].GetData(dateLine3).GetDate

**EndWith**

N🡸n+1

**EndIf**

**EndFor**

**EndFor**

#Gets the amount of invalid stock correlation number

**For integer** k 🡸 0 **to** Len(AllCorrStocks)-1

**if** AllCorrStocks[k].CorrValue=2 **then**

**integer** InvalidCount🡸InvalidCount+1

**Endif**

**EndFor**

**#**Get rid of all the empty places in the array where correlations didn’t happen

Len(AllCorrStocks)🡸 Len(AllCorrStocks)-InvalidCount

#Sort the list into an order from highest positive to highest negative.

**Function** PerformMerge(0, recordLength-1)

**EndFunction**



This algorithm calls and shows the ordered value of the correlation coefficients in a memo. Then it also gives a recommended portfolio, sized based on a user’s input which, handles any error that it may come across. The recommended portfolio is based on the stocks with the highest inter-stock correlation coefficient. It then display this on a different stock as well.

Steps:

1. Call the function AICorrBttnClick.
2. Then display the necessary data on the memo, in order of highest positive stock correlation to highest negative stock correlation.
3. Then the program validates the users input for the amount of stocks should be in the suggested portfolio.
4. Then the program adds all the stocks with the highest correlation coefficient to an array while checking that they haven’t been added before, up to the amount that the user desired.
5. It then display this in a memo box.



This algorithm is the function that is called to start the merge sort of a set of data. It is a recursive algorithm, and it splits up the data in half continuously until, there is only one element so that it can start re-assembling the data in an ordered way.

Steps:

1. If the lower value passed into the function is less than the higher value passed into the function, then.
   1. Set the middle variable as the midpoint index between the 2, values(indexes).
   2. Recursively calls the function PerformMerge on the bottom half of the data, now replacing the high values with the midpoint.
   3. Recursively calls the function PerformMerge on the top half of the data, with the low being replaced by the midpoint +1.
   4. Call the function Merge on the data, merging 2 separate data sets in orders. Continuously until the data set is sorted.

***Pseudocode:***

**Function**

PerformMerge(**integer** low, high)

#Recursively calls half of the array sizes each time splitting up element to just one element the combines again using merge function at the end, to combine in order.

**if** low<high **then**

**integer** mid🡸 (low+high) \ 2

**function** PerformMerge(low, mid)

**function** PerformMerge((mid+1), high)

**Function** Merge(low, mid, high)

**Endif**

**EndFunction**



This algorithm is the function that actually combines the data sets in order allowing the whole data set to end up ordered at the end, this function is called by the procedure PerformMerge.

Steps:

1. Sets the length of all temporary arrays, so that they can hold values if/ and when needed.
2. Iterate through from the passed in low to the mid value, doing the following.
   1. Store all the values the value from the low to the mid, of the data set in the temporary arrays indexing from 0 up.
3. Iterating through each half section of the list, do the following
   1. Compare the correlation coefficient in the temporary array and the mid value of the pulled in data set.
   2. If the one in the temporary is lower then assign the value in the temporary correlation coefficient and all its constituents to the main array, in the correct index
      1. Incrementing the low value and the original indexing variables.
   3. If the one in the temporary array is higher or equal then set main array low value index to the main array mid +1 index.
      1. Incrementing the low and the mid+1 indexing variables.
4. Finally iterates through from low to mid value, assigning the main array the values in the temp array indexed from 0, whereas the main array is indexed from low to mid.

***Pseudocode:***

**Function**

Merge(**integer** low, mid, high)

#Set the lengths all the temporary arrays used for switching

Len(temp) 🡸 Len(AllCorrStocks)

Len (tempInt1)🡸 Len(AllCorrStocks)

Len(tempInt2)🡸 Len(AllCorrStocks)

Len(tempDate1)🡸 Len(AllCorrStocks)

Len(tempDate2)🡸 Len(AllCorrStocks)

#Initialise I which determines where to store the values in the temp array

I🡸 0

#Set the the values from the passed in low to mid to the temp array, incrementing i

**For** **integer** j 🡸 low **to** mid

**Real** temp[i]🡸 AllCorrStocks[j].CorrValue

**Integer** tempInt1[i]🡸 AllCorrStocks[j].Stock1

**Integer** tempInt2[i]🡸 AllCorrStocks[j].Stock2

**String** tempDate1[i]🡸 AllCorrStocks[j].startDate

**String** tempDate2[i]🡸 AllCorrStocks[j].endDate

**integer** i🡸i+1

**EndFor**

#Set all the values of the variables so they can be used further on

**Integer** i🡸0

**Integer** j🡸 mid+1

**Integer** k🡸 low

#Checking of values and where the switching occurs between values

**While** (k<j) AND (j<=high) **do**

**if** temp[i] < AllCorrStocks[j].CorrValue **then**

**real** AllCorrStocks[k].CorrValue🡸 temp[i]

**integer** AllCorrStocks[k].Stock1🡸 tempInt1[i]

**integer** AllCorrStocks[k].Stock2🡸 tempInt2[i]

**string** AllCorrStocks[k].startDate🡸 tempDate1[i]

**string** AllCorrStocks[k].endDate🡸 tempDate2[i]

**Integer** I🡸i+1

**Integer** K🡸k+1

**Else**

**Real** AllCorrStocks[k].CorrValue🡸 AllCorrStocks[j].CorrValue

**Integer** AllCorrStocks[k].Stock1🡸 AllCorrStocks[j].Stock1

**Integer** AllCorrStocks[k].Stock2🡸 AllCorrStocks[j].Stock2

**String** AllCorrStocks[k].startDate🡸 AllCorrStocks[j].startDate

**string** AllCorrStocks[k].endDate🡸 AllCorrStocks[j].endDate

**Integer** K🡸k+1

**integer** j🡸j+1

**Endif**

**EndWhile**

#Setting the ordered array back into the main array

**For** m 🡸 k **to** j-1

**Real** AllCorrStocks[m].CorrValue🡸 temp[i]

**Integer** AllCorrStocks[m].Stock1🡸 tempInt1[i]

**Integer** AllCorrStocks[m].Stock2🡸 tempInt2[i]

**String** AllCorrStocks[m].startDate🡸 tempDate1[i]

**String** AllCorrStocks[m].endDate🡸 tempDate2[i]

**Integer** i🡸i+1

**EndFor**

**EndFunction**



This is the algorithm that executes the simple moving day average and decides whether a position of a stock is in a buy position. It also displays the buy positions of a stock in a memo. It makes the long term and short term moving day average and populates all the values in an array of an array of an array. The first level of the array is the stock, then 2nd tier of the array is the high and low of one stock, and finally the 3rd tier of the array is the actual values of the averages.

Steps:

1. Clear the memo.
2. Get the amount of populated stocks by calling the function CheckPopStocks.
3. Set the length of the 1st tier array that will store all averages, to the amount of populated stocks.
4. Sets the length of whether a stock will go up or down array to the length of populated stocks.
5. Set the high period as 27 and the low as 9.
6. Set the length of all the 2nd tier arrays to 2, for high and low.
7. Get the high and low averages of the stocks storing them in the array.
8. Iterate through every stock, doing the following.
   1. Set the length of the 2nd tiered array in the up or down array to the length of the stock minus the higher value.
   2. Iterate through each index of up or down array adding true if the smaller period average is higher than the higher period average, and false if not.
9. Add the buy dates to the memo.
   1. By iterating through each of stock.
      1. And each of the values in the stock.
      2. Checking if the smaller period is pulling the higher period average up.
      3. Finds the start of what was described above and sets one test variable to true.
      4. Find the end of this and sets the other test variable to true.
      5. When both test variable are true and the difference between the dates is greater than 5.
         1. Add the start and end date found of the buy positions to the memo.
         2. Then reset both test variables and find another start and end date where this occurs in the stock.
   2. When reached the end of the stock and all are found then do for next stock.

***Pseudocode:***

**Function**

ListAvgBttnClick()

#Clear output, and retrieve amount of stock data there is from a function

BuySellMemo.Output🡸Empty

**Boolean** periodCheck🡸True

**Integer** popStocks🡸 **function** CheckPopStock()

#Set arrays with the amount of populated stocks

Len(MyStockAverage)🡸popStocks

Len(UPorDown)🡸 popStocks

#Uses these as the 2 period values for the moving average.

**Try**

**Integer** period1🡸 **string\_to\_integer**(DayMovingValueEdit.Input)

**Integer** period2🡸 **string\_to\_integer**(MovingValue2Edit.Input)

**OnError**

OUTPUT(‘Enter valid integer value’)

**Boolean** periodCheck🡸 False

**EndTry**

#Set the length of all the 2nd tier array to 2 for the fast average and the slow average

**If** period<1 **or** period2<1 **then**

**Boolean** periodCheck🡸False

**EndIf**

**If** periodCheck=True **then**

**For integer** i 🡸 0 **to** popStocks-1

Len(MyStockAverage[i]🡸 2

**EndFor**

#Make the fast and slow average for every stock

**For** i 🡸 0 **to** popStocks-1

**Procedure** MakeAverageList(period1, i, StockIndexValues[i], 0)

**Procedure** MakeAverageList(period2, i, StockIndexValues[i], 1)

**EndFor**

**#**set the difference between the fast and slow period.

**Integer** K🡸 period2-period1

#Go through every stocks fast and slow average and

#see if the fast is bringing slow up

#If it was set boolean value to an array

**For integer** i 🡸 0 **to** popStocks-1

Len(UporDown[i])🡸Len(MyPulledStocks[StockIndexValues[i]])-

period2-1

**For integer** j 🡸 0 **to** Len(UporDown[i])-1

#bringing slow average up and general average up

**If** MyStockAverage[i][0][j+k]>MyStockAverage[i][1][j] **then**

**boolean** UporDown[i][j]🡸 True

#bringing slow average down

**Else**

**Boolean** UporDown[i][j]🡸 False

**Endif**

**EndFor**

**EndFor**

BuySellMemo.Output('Stocks Buy and Sell Times')

#Finding the dates where the stock slow average is being pulled up

# from fast average

**For** **integer** i 🡸 0 to Len (UporDown)-1

BuySellMemo.Output(MyFileNames[StockIndexValues[i]])

#Initialise variables for date finding

**Boolean** test1🡸 False

**Boolean** test2🡸 False

**#**Go through created array to see if in time frame stock is going up

# or down

**For** **integer** j 🡸 0 **to** Len(UporDown[i])-2

#Checks for changes from being brought down to now up,

set values

**If** (UporDown[i][j]=False) **AND** (UporDown[i][j+1]=True) **then**

**Integer** date1🡸 j+1

**boolean** test1🡸 True

**Endif**

**#**If the stocks starting date starts with stock going up

**If** (UporDown[i][0]=True) **AND**(UporDown[i][j+1]=True)

**AND**(j=0) **then**

**Integer** date1🡸 j+1

**boolean** test1🡸 True

**Endif**

**#**Checks for changes from the stock being brought up to down

**If** (UporDown[i][j]=True) **AND** (UporDown[i][j+1]=False) **then**

**Integer** date2🡸 j

**Boolean** test2🡸 True

**Endif**

**#**Checks if on the last date the stock is still being brought up.

**If** UporDown[i][j]=True **and** UporDown[i][j-1] **and**

j=Len(UporDown[i])-2 **then**

**integer** date2🡸 j

**boolean** test2🡸 True

**Endif**

**#**When the program finds a set of dates

# where the stock is being brought up

# for more than 5 days, display the dates

**If** (test1=True) **and** (test2=True) **then**

I**f** (date2-date1)>5 **then**

BuySellMemo.Output('Buy '+

MyPulledStocks[StockIndexValues[i]].GetData(date1).GetDate+

' - '+ MyPulledStocks[StockIndexValues[i]].GetData(date2).GetDate)

**Endif**

**#**Re-set variables

**Boolean** test1🡸 False

**Boolean** test2🡸 False

**EndIf**

**EndFor**

**EndFor**

**Else**

OUTPUT(Enter Valid integer x>0)

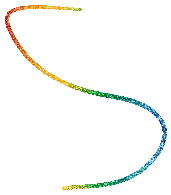
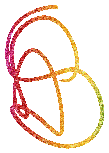
**Endif**

**EndFunction**

**User Interface:**

**A screenshot of a computer

Description automatically generated**

Figure 1

The first thing the user sees when the program starts is a pop-up dialog messages, saying the “Program Started”. Clicking OK then allows the program to proceed to the actual interface of the program. The reason for this is because it allows one to restart the program and start the program noticeably for the user so that the user is fully aware of the program starting or restarting, depending on what the user is doing. Pressing the x will still continue the program the user just needs to exit the message to continue to the actual program.

A screenshot of a computer screen

Description automatically generated

1

5

4

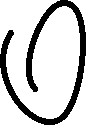
3

2

6

Figure 2

Table for figure 2 below identifying all elements.



|  |  |  |  |
| --- | --- | --- | --- |
| Number | Components / Variable | Variable Type | Description |
| 1 | IntroMemo | TMemo | This is a dynamically created memo, which is used as an introduction to the program, and as a basic set of instructions for someone to use if they don’t know what to do initially on the program. Gives a general description of what the program is meant for as well. As this is dynamically created object its all non-default properties have to be stated specifically, and it has to be parented. This memo box was made a read only memo because the user has no need to change it. |
| 2 | MaxQuantLabel | TLabel | This object is dynamically created label. It is used to show the user what the edit box beside it is for. Gives some sort of identification for the edit box. |
| 3 | MaxDataEdit | TEdit | This is a dynamically created object that is used to set the maximum amount of stocks that can be had in one instance. The default number is set to 5. However it can be changed. The minimum stocks possible is 5. And the edit box is handled so that the user can only enter an integer. If they enter a integer under 5 the max stock is set to 5 and the edit box is changed back to 5. |
| 4 | RestartBttn | TButton | This is a dynamically created button that is used to restart the program, it re-sets all variables, arrays and components. It allows the user to start again without having to exit the program and start again. |
| 5 | MaxDataBttn | TButton | This is a dynamically created button, that is used to actually set the max data values that can be stored. It takes the data in the MaxDataEdit component and uses it to set the maximum stocks that can be stored at one time. It is where the error checking also occurs. Also sets the array length of the file names, averages, and the index values of the stocks. |
| 6 | MainMenu1 | TMainMenu | This is a dynamically created main menu. Which stores all the items of the main menu. |
| 6 | MainMenu2 | TMenuItem | This is a dynamically created menu item that can be clicked so to change to the different sections of the programs. This is used to make different tabs of the program that can be switched to at the discretion of the user. Clicking this shows all the menu items that show the user the different sections depending on which menu item the user clicked. |

A screenshot of a computer

Description automatically generated

1

2

3

4

5

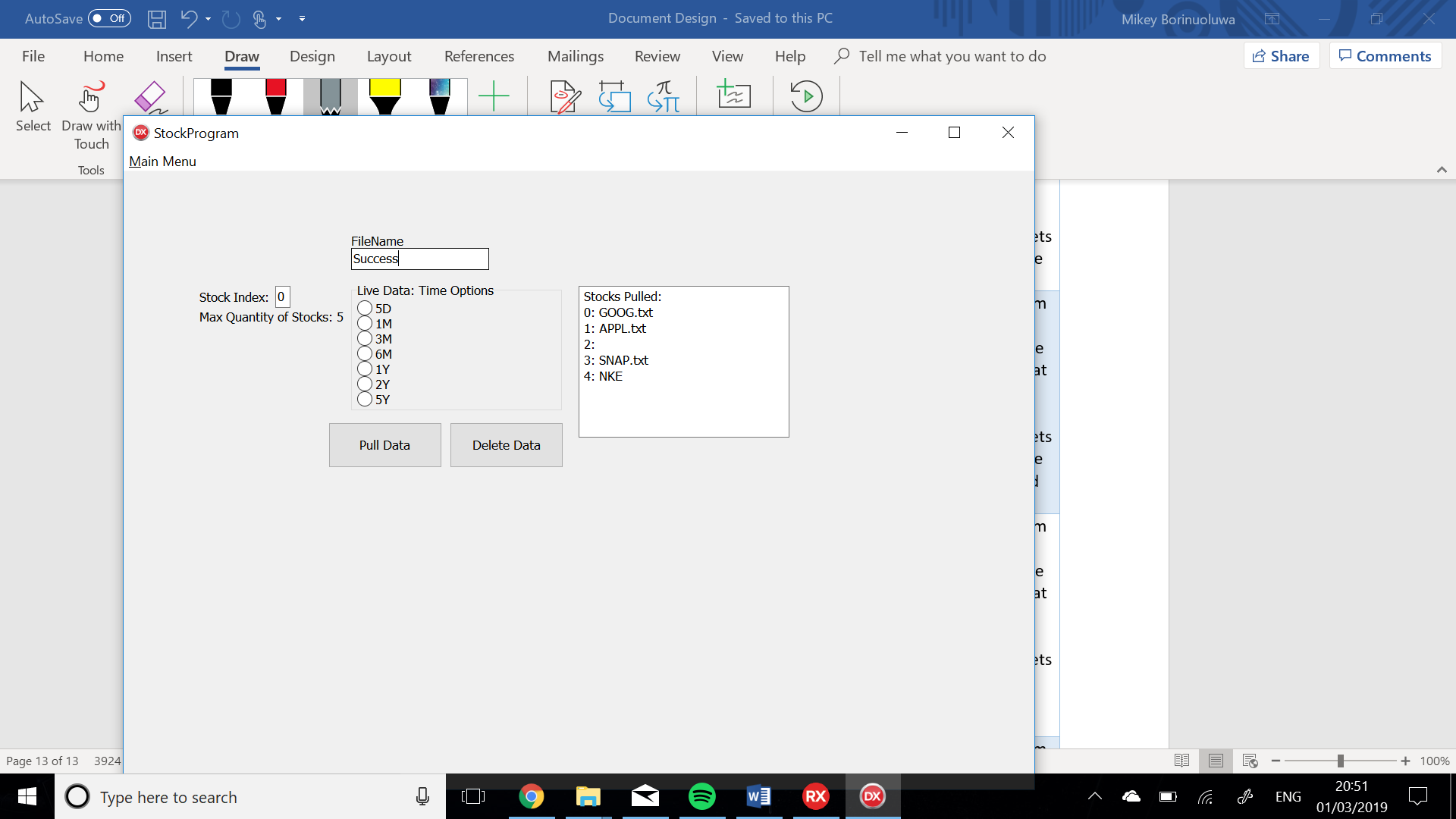
6

7

Figure 3

Table for figure 3 identifying all elements found below.

|  |  |  |  |
| --- | --- | --- | --- |
| Number | Components/Variable | Variable Type | Description |
| 1 | New1 | TMenuItem | This is a dynamically created menu item that when clicked takes you to the section that is described. It hides all the components that aren’t needed for that sections and shows the ones that are needed in their correct positions and sizes. This particular menu item also sets the text in the intro, the caption in the label, the caption in the button, and the initial text in the edit. It also sets the property of the memo to a read only. |
| 2 | RetrieveData1 | TMenuItem | This is a dynamically created menu item that when clicked takes you to the section that is described. It hides all the components that aren’t needed for that sections and shows the ones that are needed in their correct positions and sizes. This particular menu item also sets the text in the stocks pulled memo, the caption in all the labes, the captions for all the buttons, the initial text in all the edits, and the title of the radio group. |
| 3 | Graphing1 | TMenuItem | This is a dynamically created menu item that when clicked takes you to the section that is described. It hides all the components that aren’t needed for that sections and shows the ones that are needed in their correct positions and sizes. This particular menu item also sets the caption in the button, and sets the graph to a 2D viewing, and sets the property of read only on the memo and display to true. |
| 4 | Display1 | TMenuItem | This is a dynamically created menu item that when clicked takes you to the section that is described. It hides all the components that aren’t needed for that sections and shows the ones that are needed in their correct positions and sizes. This particular menu item also sets the caption in the buttons, and sets the display to a read only. |
| 5 | Analysis | TMenuItem | This is a dynamically created menu item that when clicked takes you to the section that is described. It hides all the components that aren’t needed for that sections and shows the ones that are needed in their correct positions and sizes. This particular menu item also sets the text in the edits, the captions in the labels, the captions in the buttons, and sets the display as read only. |
| 6 | AI | TMenuItem | This is a dynamically created menu item that when clicked takes you to the section that is described. It hides all the components that aren’t needed for that sections and shows the ones that are needed in their correct positions and sizes. This particular menu item also sets the caption in the button, and sets the property of both memo’s being read only to true. |
| 7 | Exit1 | TMenuItem | This is a dynamically created menu item that when clicked exits the program. |



14

13

12

11

10

2

9

8

7

6

5

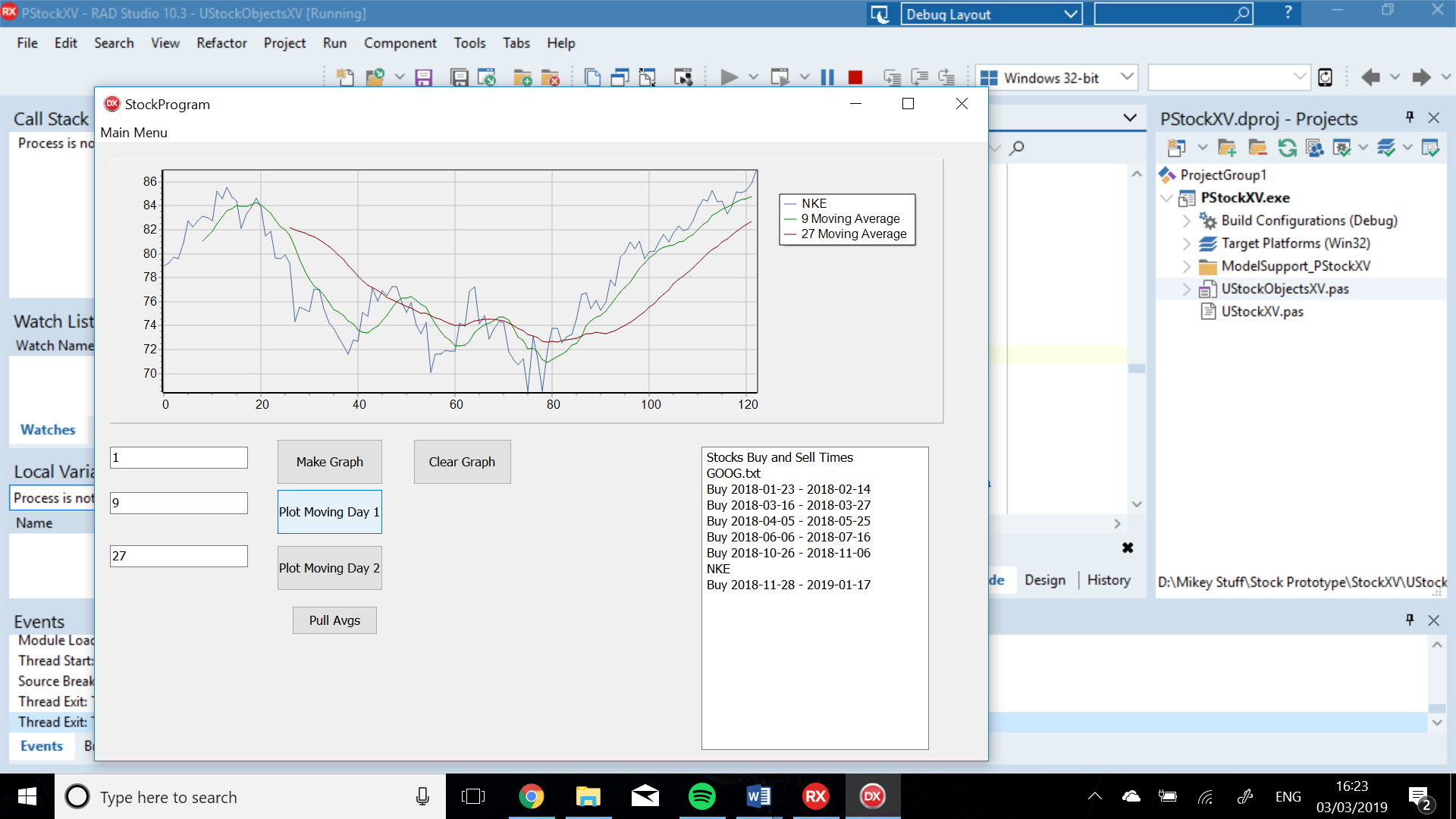
4

3

1

Figure 4

|  |  |  |  |
| --- | --- | --- | --- |
| No. | Component/Variable | Variable Type | Description |
| 1 | MainMenu2 | TMenuItem | This is a dynamically created menu item that can be clicked so to change to the different sections of the programs. This is used to make different tabs of the program that can be switched to at the discretion of the user. Clicking this shows all the menu items that show the user the different sections depending on which menu item the user clicked. |
| 2 | QuantPullEdit | TEdit | This is a dynamically created object that is used to index where the data should be pulled to or deleted from in the variable MyPulledStocks. The default number is set to 0. The edit increments every time the ‘Pull Data’ button is clicked. The value is used to index to that place in the array, when saving the new data or deleting the old data. When it reaches the maximum amount of stocks required the edit wraps around and becomes 0 again. If a stock wants to be deleted they use this to tell the program which one they want to delete by entering the index of the stock that they want to be deleted. |
| 3 | IndexLabel | TLabel | This object is dynamically created label. It is used to show the user what the edit box beside it is for. Gives some sort of identification for the edit box, in its caption, and positioned accordingly. |
| 4 | MaxQuantLabel | TLabel | This object is dynamically created label. It is used to show the user what the edit box beside it is for. Gives some sort of identification for the edit box, in its caption, and positioned accordingly. |
| 5 | FileNameLabel | TLabel | This object is dynamically created label. It is used to show the user what the edit box under it is for. Gives some sort of identification for the edit box, in its caption, and positioned accordingly. |
| 6 | StockNameEdit | TEdit | This is a dynamically created objected that is used to find the stock data. There are 2 options for the stock to enter in a file, or a ticker name for the stocks. If not a file is entered the program uses the ticker symbol to try find the stock from the API, using the names as a parameter. This edit box is also used just after a stock has been pulled. It shows some handled errors, such as telling the user pulling the data was a success by showing success, or if the file can’t be found. |
| 7 | RadioGroup | TRadioGroup | This is a dynamically created radio group. It is used to determine the time scale that should be received of the stock requested. Added to the URL that is trying to be pulled. Depending on which radio item is clicked. This is not used for a .txt file as the data is already pulled. |
| 8 | (Radio Group Items) | (Items of Radio Group) | These are the dynamically created items that is in the radio group. Only one of these can be clicked as only one is needed for the API when pulled. The names of the items are ordered accordingly, D in replace of days, M in replace of months, and Y in replace for years. |
| 9 | PullDatBttn | TButton | This is a dynamically created button that is used to pull the data of the program, into the MyPulledStocks variable, it uses the IndexLabel to find where to place it. This then initially tries to find the file in the debug directory. If it can find the file it calls the creation of TStock and places it in the correct index of the MyPulledStocks. If it can’t find it, the program then switches to searching on the API, it uses the API, the input if the file name from the edit and the radio group items corresponding values. If it can finds it then creates the object TStock object with the API, and inputs it in the correct index of MyPulledStocks. It also handles the error if the name entered isn’t a valid URL by showing a message. |
| 10 | DeleteDataBttn | TButton | This is a dynamically created button, that is used to delete a stock that is no longer needed by the user. It uses the QuantPullEdit to know which stock to delete. It sets that value in the array to nil, overwriting all original data. |
| 11 | (Example of .txt Stock) | (MyPulledStocks) | Here you can see a stock that has been successfully pulled from a .txt file and its name has been added to the StockPulledMemo memo. You can differentiate the API stocks from the .txt files by simply checking if the name contains a .txt at the end. |
| 12 | StockPulledMemo | TMemo | This is a dynamically created memo, which is used to shows the user all the pulled stocks that they currently have pulled in this run-time. Shows both stocks gotten from API or .txt. |
| 13 | (Example of a deleted Stocks) | Nil | Here you can see a stock that has been successfully deleted its name has been removed from the StockPulledMemo memo. The index of array MyPulledStocks will go to nil |
| 14 | (Example of a stock from API) | (MyPulledStocks) | Here you can see a stock that a stock from the API has been successfully pulled, and its name has been added to the StockPulledMemo memo. You can differentiate the API stocks from the .txt files by simply checking if the name contains a .txt at the end. |



14

12

11

1

2

3

4

6

5

7

10

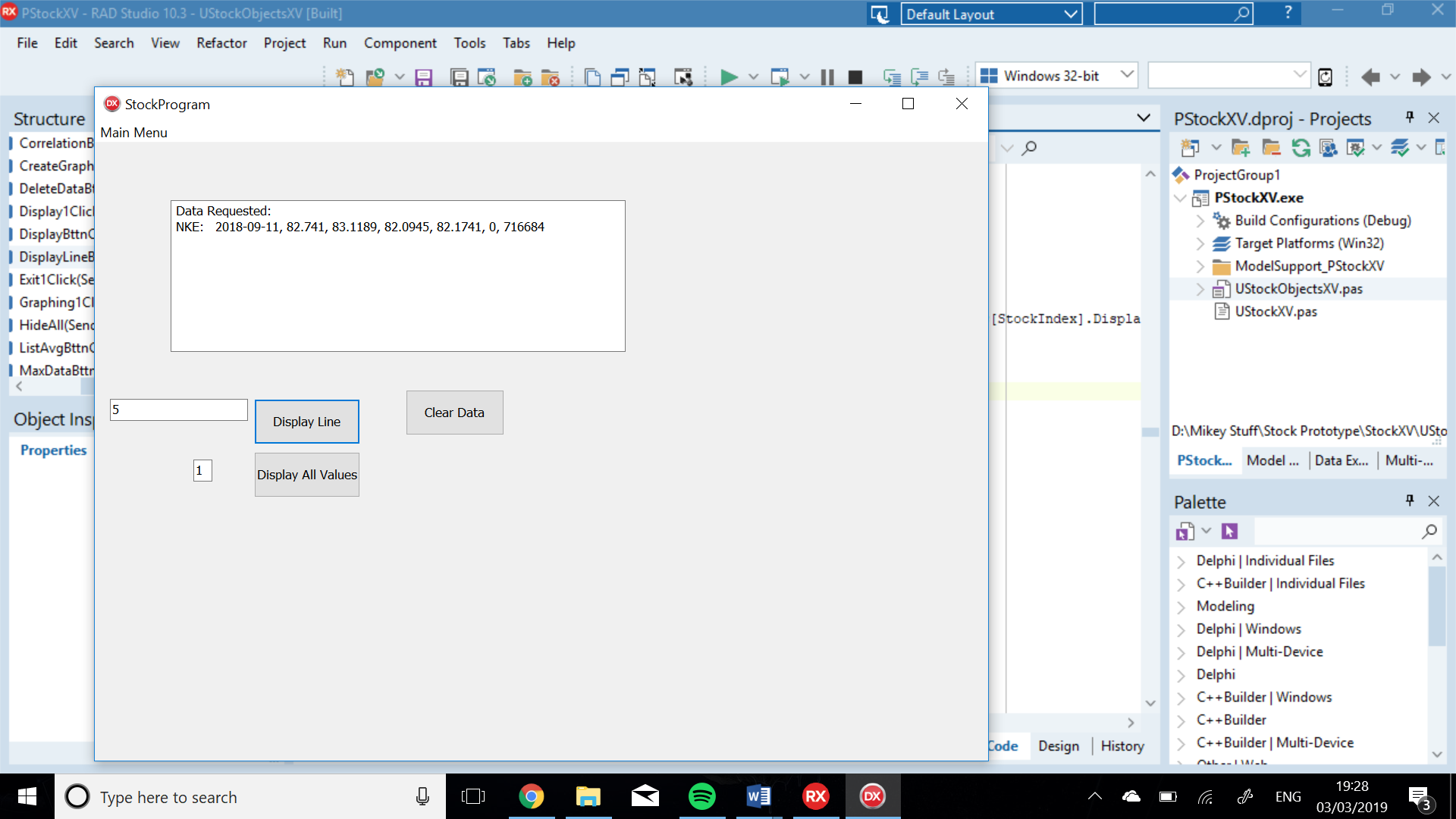
8

9

13

Figure 5

|  |  |  |  |
| --- | --- | --- | --- |
| No. | Component / Variable Name | Variable Type | Description |
| 1 | Graph | TChart | This is a dynamically created chart. It Is made to plot all the data of the stocks, to make it easy to see the trend lines that are being made by the stock. You can see the legend on the top right of the chart. |
| 2 | StockSeries1 | TFastLineSeries | This is a dynamically created TFastLineSeries, its parent chart is Graph. This allows this series to appear on this chart. The reason that a fast line series was chosen is because it could be plotting a lot of data, and when there is a lot of data coming in, and time is of the essence this variable type was more favoured than TLineSeries. This fast line series takes all the data points and plots them on the chart. |
| 3 | MovingSeries | TFastLineSeries | This is a dynamically created TFastLineSeries, its parent chart is Graph. This allows this series to appear on this chart. The reason that a fast line series was chosen is because it could be plotting a lot of data, and when there is a lot of data coming in, and time is of the essence this variable type was more favoured than TLineSeries. This fast line series calculates a moving day average of the stock, with the day period being set by the edit box on its left. |
| 4 | (Legend of Graph) | (Legend) | This is the legend that is part of the dynamically created, chart. It tells the user to which series is which. |
| 5 | StockIndexEdit | TEdit | This is a dynamically created TEdit object. This edit box is used to determine which stock to plot on the chart. It takes any value, however when the button to plot is clicked the chart will only plot a valid integer that refers to the index of stock you want to plot. |
| 6 | MovingSeries2 | TFastLineSeries | This is a dynamically created TFastLineSeries, its parent chart is Graph. This allows this series to appear on this chart. The reason that a fast line series was chosen is because it could be plotting a lot of data, and when there is a lot of data coming in, and time is of the essence this variable type was more favoured than TLineSeries. This fast line series calculates a moving day average of its stock, with the day period being set by the edit box on its left. |
| 7 | DayMovingValueEdit | TEdit | This is dynamically created TEdit object. This is an edit box which is used to determine the period of the first moving day average. It takes an integer value greater than 0. Which is then used when the button to its right is clicked, edit for MovingSeries1. |
| 8 | CreateGraphBttn | TButton | This is a dynamically created TButton object. This button is uses the StockIndexEdit, to know which graph to plot. It then plots the data on the StockSeries series which is on the chart Graph. Also checks that StockIndexEdit is a valid integer. |
| 9 | ClearGraphBttn | TButton | This is a dynamically created TButton. That is used to clear all the graph’s content. It can be used to restart the graphing process. |
| 10 | MovingDayAverage1Bttn | TButton | This is a dynamically created TButton. That is used to actually plot the moving day value. Using the value given in the DayMovingValueEdit as the period over what to make the average. It uses MovingSeries1, as the series on Graph to plot the data. It also error checks the value in DayMovingValueEdit checking that is greater than 0 and an integer. |
| 11 | MovingValue2Edit | TEdit | This is dynamically created TEdit object. This is an edit box which is used to determine the period of the first moving day average. It takes an integer value greater than 0. Which is then used when the button to its right is clicked, for MovingSeries2. |
| 12 | MovingAverage2Bttn | TButton | This is a dynamically created TButton. That is used to actually plot the moving day value. Using the value given in the MovingValue2Edit as the period over what to make the average. It uses MovingSeries2, as the series on Graph to plot the data. It also error checks the value in MovingValue2Edit checking that is greater than 0 and an integer. |
| 13 | BuySellMemo | TMemo | This is a dynamically created TMemo object. It has been given the property that it is read only. The memo is populated when the button ‘Pull Avgs’ is clicked. It shows the user for all the stock that they pulled when were the times to buy this stock as it would have been profitable. |
| 14 | ListAvgBttn | TButton | This is a dynamically created TButton object. This button is used to list all the positios where the user should have bought each stock. It uses the simple moving day average algorithm to work this out. |



1

4

2

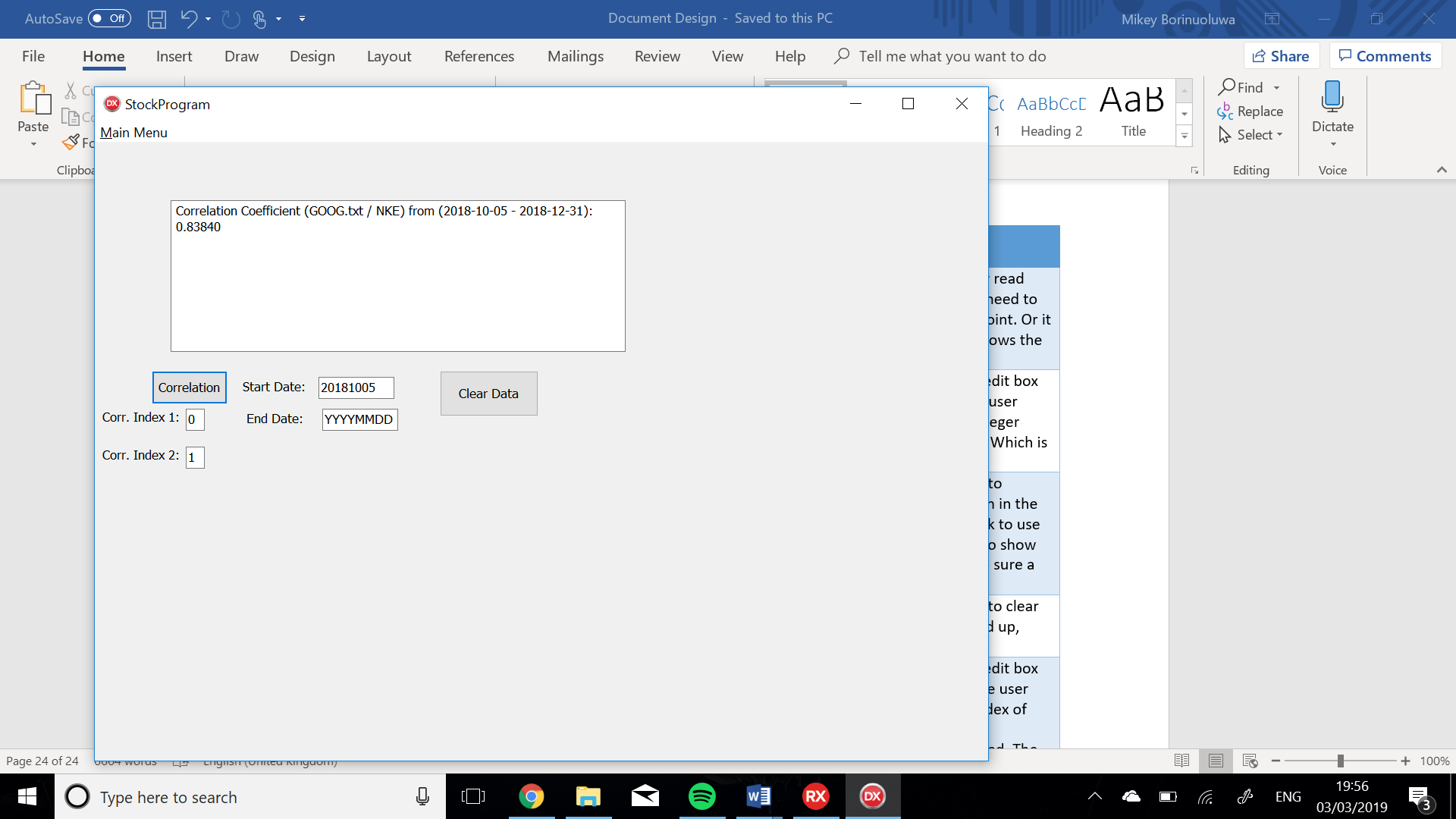
3

6

5

Figure 6

|  |  |  |  |
| --- | --- | --- | --- |
| No. | Component/ Variable Name | Variable Type | Description |
| 1 | DisplayAllMemo | TMemo | This is a dynamically created TMemo. The property read only of this memo is set to true as the user has no need to edit it. This shows all the values for a single data point. Or it can show all the datapoints and all the values. It allows the user to see the actual value of a data point. It shows the user the stock which is relevant, and all the relevant values. |
| 2 | LineValueEdit | TEdit | This is dynamically created TEdit object. This is an edit box which is used to determine the which line that the user wants to display of the chosen stock. It takes an integer value between the first and last index of the stock. Which is then used when the button to its right is clicked. |
| 3 | DisplayLineBttn | TButton | This is a dynamically created TButton. That is used to actually display the line value. Using the value given in the lineValueEdit and the StockIndexEdit as which stock to use and which line to get. It uses the DisplayAllMemo to show the line, it also error checks the edit boxes to make sure a valid input is taken. |
| 4 | ClearDataBttn | TButton | This is a dynamically created TButton. That is used to clear the DisplayAllMemo. If the memo is getting clogged up, visually. |
| 5 | StockIndexEdit | TEdit | This is dynamically created TEdit object. This is an edit box which is used to determine the which stock that the user wants to display. It takes an integer value of the index of the stock Which is then used when the either the DisplayLineBttn is clicked or the DisplayBttn is clicked. The value of this edit is also error checked to be valid. |
| 6 | DisplayBttn | TButton | This is a dynamically created TButton. That is used to actually display the all the line values of the desired stock. Using the value given in the StockIndexEdit as which stock to use. It uses the DisplayAllMemo to show the stock, it also error checks the edit boxes to make sure a valid input is taken. |



1

3

2

4

5

6

7

8

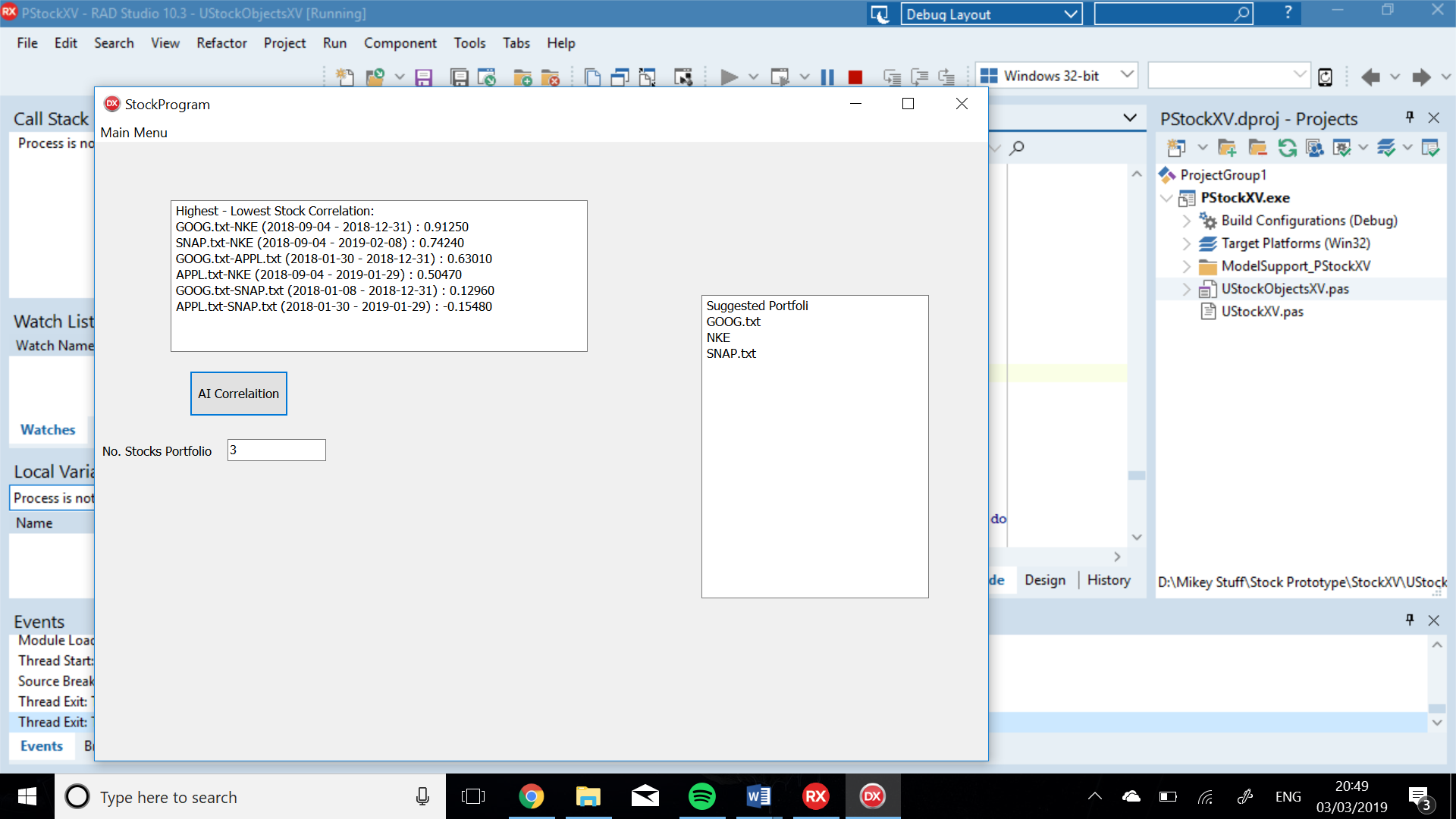
9

10

11

Figure 7

|  |  |  |  |
| --- | --- | --- | --- |
| No. | Component/ Variable | Variable Type | Description |
| 1 | DisplayAllMemo | TMemo | This is a dynamically created TMemo. The property read only of this memo is set to true as the user has no need to edit it. This shows the correlation coefficient between the 2 stocks, it states the 2 stocks and it states the dates the correlation coefficient was taken between. |
| 2 | CorrIndex1Label | TLabel | This object is dynamically created label. It is used to show the user what the edit box under it is for. Gives some sort of identification for the edit box, in its caption, and positioned and captioned accordingly. |
| 3 | CorrelationBttn | TButton | This is a dynamically created TButton. That is used to actually display and calculate the correlation coefficient between the 2 stocks. It uses the values given in the edit boxes for the stocks to correlate and the dates to take the correlation between(optional). It also error checks the edit boxes to make sure a valid input is taken. |
| 4 | StartCorrLabel | TLabel | This object is dynamically created label. It is used to show the user what the edit box under it is for. Gives some sort of identification for the edit box, in its caption, and positioned and captioned accordingly. |
| 5 | DateCorrStartEdit | TEdit | This is dynamically created TEdit object. This is an edit box which is used to determine the starting date that the correlation should start at that is desired by the user. The entered text is attempted to be found in both stocks. The date has to be formatted specifically in this format: YYYYMMDD. Which is then used when the CorrelationBttn is clicked. If the text is left as YYYYMMDD the program finds the date that allows the greatest time period that is possible between the 2 stocks. The value of this edit is also error checked to be valid. |
| 6 | CorrIndex2Label | TLabel | This object is dynamically created label. It is used to show the user what the edit box under it is for. Gives some sort of identification for the edit box, in its caption, and positioned and captioned accordingly. |
| 7 | StockCorr2Edit | TEdit | This is dynamically created TEdit object. This is an edit box which is used to determine the which stock that the user wants to use as one of the correlating stocks. It takes an integer value of the index of the stock Which is then used when the CorrelationBttn is clicked. The value of this edit is also error checked to be valid. |
| 8 | StockCorr1Edit | TEdit | This is dynamically created TEdit object. This is an edit box which is used to determine the which stock that the user wants to use as one of the correlating stocks. It takes an integer value of the index of the stock Which is then used when the CorrelationBttn is clicked. The value of this edit is also error checked to be valid. |
| 9 | EndCorrLabel | TLabel | This object is dynamically created label. It is used to show the user what the edit box under it is for. Gives some sort of identification for the edit box, in its caption, and positioned and captioned accordingly. |
| 10 | DateCorrEndEdit | TEdit | This is dynamically created TEdit object. This is an edit box which is used to determine the starting date that the correlation should start at that is desired by the user. The entered text is attempted to be found in both stocks. The date has to be formatted specifically in this format: YYYYMMDD. Which is then used when the CorrelationBttn is clicked. If the text is left as YYYYMMDD the program finds the date that allows the greatest time period that is possible between the 2 stocks. The value of this edit is also error checked to be valid. |
| 11 | ClearDataBttn | TButton | This is a dynamically created TButton. That is used to clear the DisplayAllMemo. If the memo is getting clogged up, visually. |



1

2

3

4

5

Figure 8

|  |  |  |  |
| --- | --- | --- | --- |
| No. | Component/ Variable | Variable Type | Description |
| 1 | DisplayAllMemo | TMemo | This is a dynamically created TMemo. The property read only of this memo is set to true as the user has no need to edit it. This shows the correlation coefficient between the all the stocks in order from highest positive to highest negative, it states the 2 stocks and it states the dates the correlation coefficient was taken between. |
| 2 | SuggestPortMemo | TMemo | This is a dynamically created TMemo. The property read only of this memo is set to true as the user has no need to edit it. This shows the suggested portfolio that the user should have had to maximise profits. The amount of stocks shown is done by the MaxPortEdit |
| 3 | ShowAiBttn | TButton | This is a dynamically created TButton. That is used to actually display and calculate the correlation coefficient between the all the stocks pulled and it sorts them into the correct order. It does the correlation based on the maximum time period possible. Furthermore it shows the suggested portfolio that will give the highest profits. It also error checks the edit boxes to make sure a valid input is taken. |
| 4 | MaxPortEdit | TEdit | This is dynamically created TEdit object. This is an edit box which is used to determine the maximum amount of stocks that the program should make a portfolio with. Which is then used when the ShowAiBttn is clicked. The value of this needs to be less than the value of the amount of stocks pulled. Additionally the value of this edit is also error checked to be valid. |
| 5 | MaxPortLabel | TLabel | This object is dynamically created label. It is used to show the user what the edit box under it is for. Gives some sort of identification for the edit box, in its caption, and positioned and captioned accordingly. |