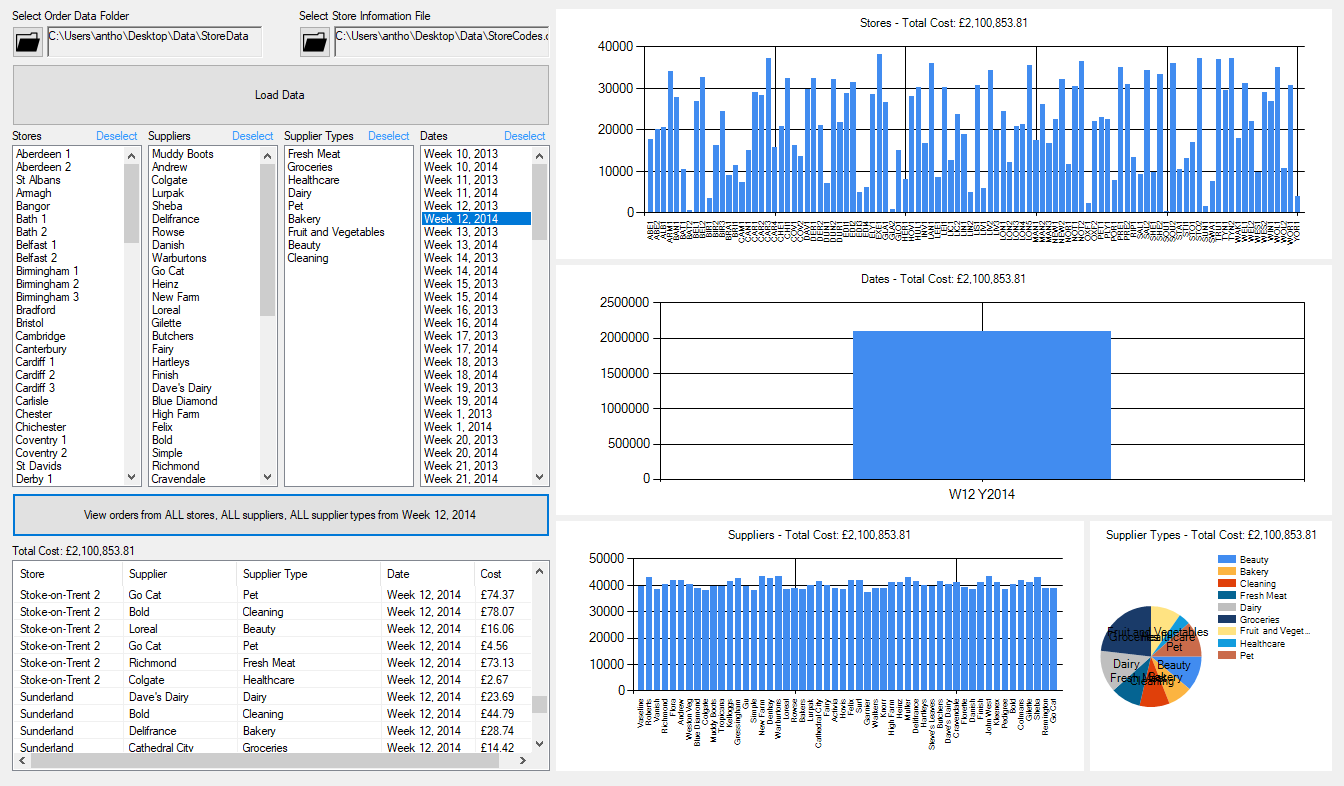
Task-Based Software Engineering Assignment Documentation

Anthony Sturdy / Staffordshire University



# Application Design

The design of the C# application was iterated upon multiple times throughout development, for a balance of speed, features and to improve usability. I wanted the application to be easy to use, and for the user to know how each feature works without having to read about it.

The initial design of the application was as follows:



This iteration was to test loading the data. The data directories were specified, then the List Boxes were populated with the information. The List Boxes are used to filter the data as the user needs to. For graphing the results, the original idea was to have another form open when the user decides to view the data. I decided against that and went with a one-window approach, as can be seen in the next iteration:



This iteration sees the addition of WinForms’ Charts on the right of the window to graph the data. When thinking about how to graph the data I was struggling to thing of a good style of chart which would allow me to present all four elements, so decided to go with four separate charts. Three of which are column bar charts, and the final is a pie chart. A pie chart would’ve better represented how the cost is split between the different options (e.g. Groceries, Dairy, Beauty, etc. Stoke-On-Trent, Brighton, Wolverhampton, etc), but with there being so many options for Stores, Suppliers and Dates, using a pie chart did not look good and was unreadable, which is why it is only used for the Supplier Type which only has a small amount of options available.

The next iteration has only a small change but improved the data loading speed drastically.

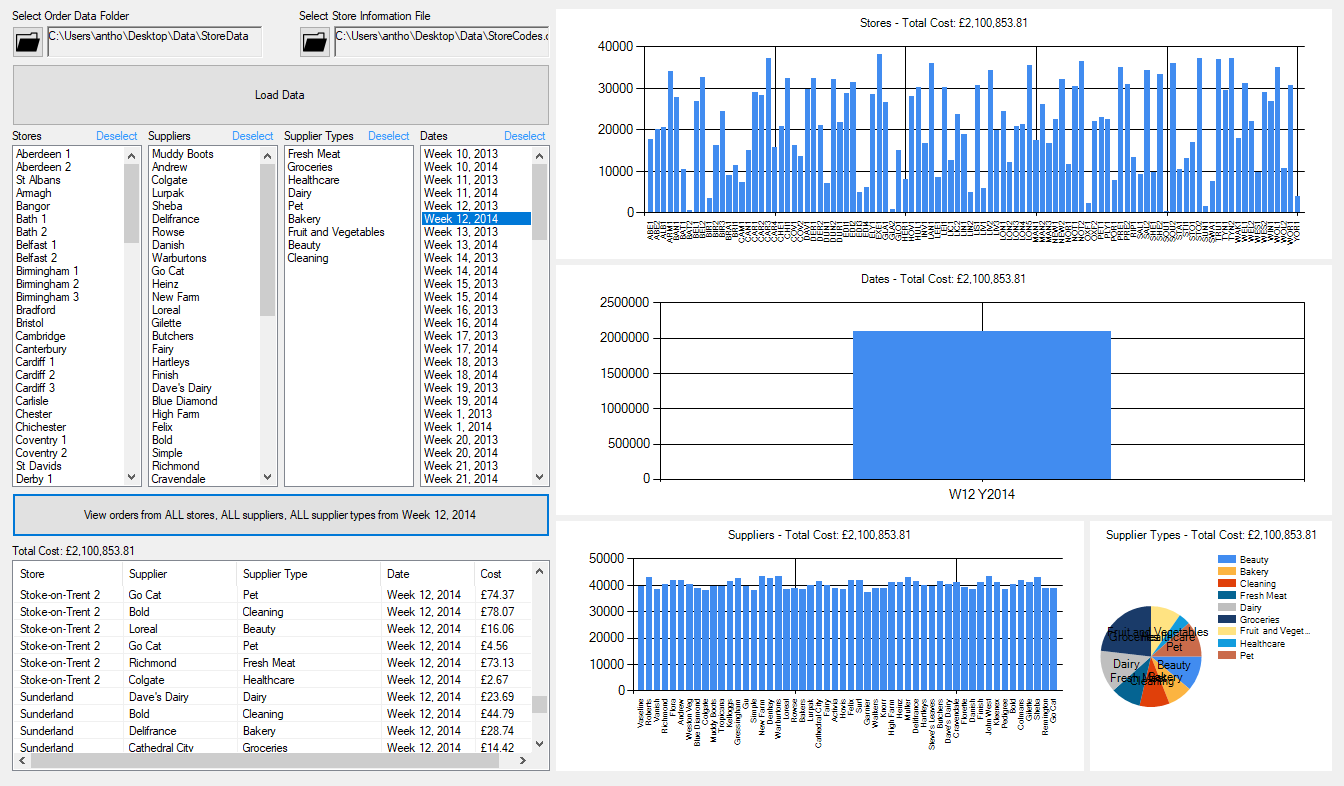


The removal of the loading bar enabled me to increase the speed of data loading from an average of 25.876 seconds to an average of 8.808 seconds, meaning the time to load the data decreased by 65.96%. The reason for this increase was that updating then refreshing a UI element for each loop caused it to slow down considerably. Although I think the loading bar was a great addition to the application, with such an increase of speed I did not think it was worth it.

Lastly, the final iteration of the application allows the user to see a List View of the filtered data alongside the charts. The charts were good to quickly see where the money spent is being split up but didn’t show individual orders for a more detailed view.



This extra feature completes the application, allowing the different filters to be applied and graphed straight away, and for each order to be individual inspected by the user. The above design can be compared to the final application:

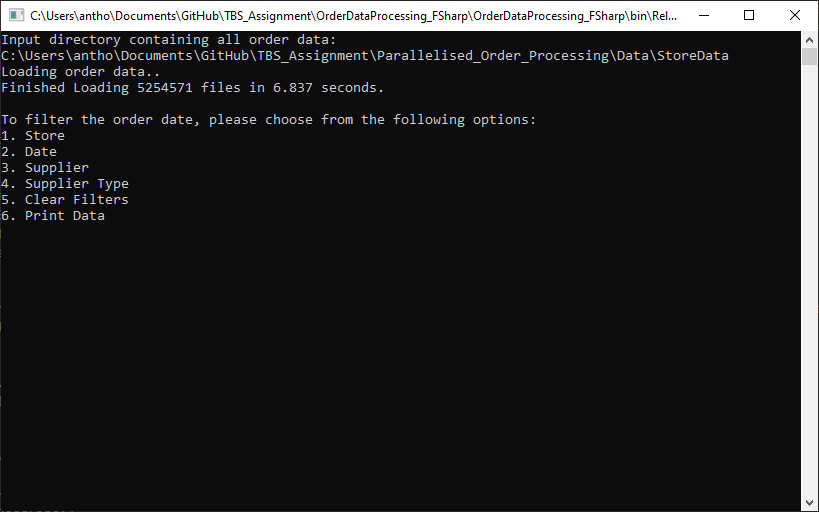


In this screenshot, the user is viewing all data from Week 12 in 2014. When different filters are selected, the ‘Show Filtered Data’ button text updates to show what filters will be applied.

# F# Implementation

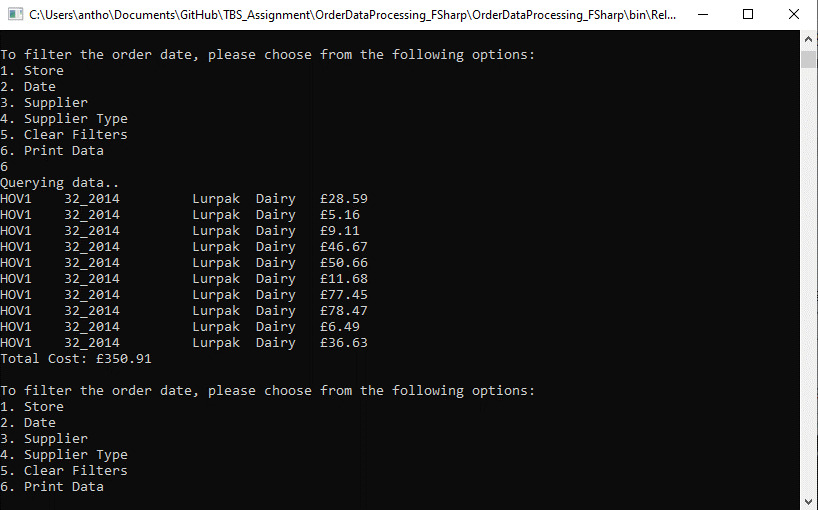
**Overview**

The F# implementation of the application is similar to the C# application, without the WinForms UI. Firstly, the user is asked to input a directory for the Order data, the Store data is not used in the F# version. Next, 6 options are displayed:



These options give the player the same control over querying the data as the C# implementation does. After each option is complete (e.g. Setting a filter for the Date), they are presented with the menu again. The user can apply different filters without having to reload the data, the filters can be cleared, and if the user types ‘quit’ the application will exit.

Option 6 prints the data with the selected filters applied, here is an example of the filtered orders which are from Brighton and Hove, Lurpak and from Week 32 of 2014:



When over there are still over 150,000 results after the filters have been applied, the user is asked whether they’d still like to continue to print the results, as it could take a while to fully print. If no filters are applied there are over 5 million results in the data set provided.

**Source Code**

The source code for the F# implementation can be found below.

open System

open System.IO

open System.Linq

open System.Diagnostics

open System.Threading.Tasks

open System.Collections.Concurrent

let mutable fileDir = ""

let mutable storeFilter = ""

let mutable dateFilter = ""

let mutable supplierFilter = ""

let mutable supplierTypeFilter = ""

type Order(*\_storeCode* : string, *\_date* : string, *\_suppName* : string, *\_suppType* : string, *\_cost* : double) =

    member x.storeCode = \_storeCode

    member x.date = \_date

    member x.suppName = \_suppName

    member x.suppType = \_suppType

    member x.cost = \_cost

let orders = new ConcurrentBag<Order>()

let Initialise() =

    while fileDir = "" do    *// Need file directory before continuing*

        Console.WriteLine("Input directory containing all order data:")

        fileDir <- Console.ReadLine()

*// Loop through all files*

    let fileNames = ConcurrentBag<string>(Directory.GetFiles(@"" + fileDir))

    Console.WriteLine("Loading order data..")

    let sw: Stopwatch = new Stopwatch()

    sw.Start()

    let p = Parallel.ForEach(fileNames, (fun*path*->

        let fileName = Path.GetFileNameWithoutExtension(path)

        let splitFileName: string[] = fileName.Split('\_')

        let store: string = splitFileName.[0]

        let date: string = (splitFileName.[1] + "\_" + splitFileName.[2])

*// Loop through each line of this file*

        let orderData: string[] = File.ReadAllLines(path)

        for orderLine in orderData do

            let splitOrderInfo: string[] = orderLine.Split(',')

            let supplierName: string = splitOrderInfo.[0]

            let supplierType: string = splitOrderInfo.[1]

            let cost: double = Convert.ToDouble(splitOrderInfo.[2])

            let order: Order = new Order(store, date, supplierName, supplierType, cost)

            orders.Add(order)

    ))

    sw.Stop()

    Console.WriteLine("Finished Loading " + Convert.ToString(orders.Count) + " files in " + Convert.ToString(Convert.ToDouble(sw.ElapsedMilliseconds)/1000.0) + " seconds.")

let PrintOptions() =

*// Print user options*

    Console.WriteLine("\nTo filter the order data, please choose from the following options:")

    Console.WriteLine("1. Store")

    Console.WriteLine("2. Date")

    Console.WriteLine("3. Supplier")

    Console.WriteLine("4. Supplier Type")

    Console.WriteLine("5. Clear Filters")

    Console.WriteLine("6. Print Data")

let PrintData() =

    Console.WriteLine("Querying data..")

    let qObject = orders.ToArray().AsParallel<Order>()

*//F# doesn't have ternary (?) operator, so this goes:*

*//If filter is empty, return true, else, return whether filter matches order details*

    let query =

        qObject

        |> Seq.filter(fun*o*-> if storeFilter = "" then true else o.storeCode = storeFilter)

        |> Seq.filter(fun*o*-> if dateFilter = "" then true else o.date = dateFilter)

        |> Seq.filter(fun*o*-> if supplierFilter = "" then true else o.suppName = supplierFilter)

        |> Seq.filter(fun*o*-> if supplierTypeFilter = "" then true else o.suppType = supplierTypeFilter)

    let totalCost =

        query

        |> Seq.sumBy( fun*o*-> o.cost )

    if query.Count() = 0 then

        Console.WriteLine("No matches found with specified filters!")

    else if query.Count() > 150000 then

        Console.WriteLine("A lot of results (" + Convert.ToString(query.Count()) + ") are about to be printed, continue? (y/n)")

        let choice = Console.ReadLine()

        if choice <> "n" then

            for o in query do

                Console.WriteLine(o.storeCode + "\t" +

                o.date + "\t\t" +

                o.suppName + "\t" +

                o.suppType + "\t£" +

                Convert.ToString(o.cost))

            Console.WriteLine("Total Cost: £" + Convert.ToString(totalCost))

    else

        for o in query do

            Console.WriteLine(o.storeCode + "\t" +

            o.date + "\t\t" +

            o.suppName + "\t" +

            o.suppType + "\t£" +

            Convert.ToString(o.cost))

        Console.WriteLine("Total Cost: £" + Convert.ToString(totalCost))

let CheckInput(*input*: string) =

    if input = "1" then

        Console.WriteLine("Enter Store code you'd like to filter: (e.g. HOV1)")

        storeFilter <- Console.ReadLine()

    else if input = "2" then

        Console.WriteLine("Enter Date you'd like to filter: (e.g. 13\_2014 would be Week 13, 2014)")

        dateFilter <- Console.ReadLine()

    else if input = "3" then

        Console.WriteLine("Enter Supplier you'd like to filter: (e.g. Lurpak)")

        supplierFilter <- Console.ReadLine()

    else if input = "4" then

        Console.WriteLine("Enter Supplier Type you'd like to filter: (e.g. Dairy)")

        supplierTypeFilter <- Console.ReadLine()

    else if input = "5" then

        Console.WriteLine("Clearing filters")

        storeFilter <- ""

        dateFilter <- ""

        supplierFilter <- ""

        supplierTypeFilter <- ""

    else if input = "6" then

        PrintData()

    PrintOptions()

let MainLoop() =

    Initialise()

    PrintOptions()

    let mutable quit = false

    while quit <> true do

        let input = Console.ReadLine()

        match input with

        | "quit" -> quit <- true

        | \_ -> CheckInput(input)

MainLoop()

**F# Compared with C#**

F# - also being a .NET language – has a lot of similarities to C#. I found that any of C#’s Standard Library functionality as available for use in F# too, which was helpful for converting the C# implementation to F#.

The F# application performance was similar to the C# performance also. Loading the data in C#, with Parallelism, took on average 6.224 seconds. Loading the data in F#, with Parallelism, took on average 6.821 seconds. In the C# implementation, Parallelism isn’t used for loading the data due to problems with it crashing inconsistently, meaning the average to load files in C# without Parallelism is 8.808 seconds.

The biggest difference between C# and F# I found was the syntax. Learning the F# syntax was also the biggest challenge I faced when converting the application. F# has some operators and syntax I’ve never used before in any language, such as the Pipe Operator (|>), Not Equal Comparison Operator is different to languages I have personally used (<>) and that all variables are immutable by default. F# doesn’t use curly brackets for defining a block of code, instead indentation is important to define blocks of code. This wasn’t completely unfamiliar to me as I have used Python previously, but it was strange after going from C# to this.

Lastly, I found F# code to be more readable and concise than C# code in some situations. For example, both implementations use PLINQ to query the data using the filters specified by the user. The C# code for this looks like:

var parallelQuery = from num in orders.AsParallel()

                    select num;

if(selectionIndices[(int)INDICES.STORE] != -1)

    parallelQuery = parallelQuery.Where(o => o.store.code == stores.Values.ElementAt(selectionIndices[(int)INDICES.STORE]).code);

if (selectionIndices[(int)INDICES.SUPPLIER] != -1)

    parallelQuery = parallelQuery.Where(o => o.supplierName == supplierNames.ElementAt(selectionIndices[(int)INDICES.SUPPLIER]));

if (selectionIndices[(int)INDICES.SUPPLIER\_T] != -1)

    parallelQuery = parallelQuery.Where(o => o.supplierType == supplierTypes.ElementAt(selectionIndices[(int)INDICES.SUPPLIER\_T]));

if (selectionIndices[(int)INDICES.DATE] != -1)

    parallelQuery = parallelQuery.Where(o => (o.date.week + "" + o.date.year) == dates.ElementAt(selectionIndices[(int)INDICES.DATE]).week + "" + dates.ElementAt(selectionIndices[(int)INDICES.DATE]).year);

var queryList = parallelQuery.ToList();

The same functionality in F# looks like:

let qObject = orders.ToArray().AsParallel<Order>()

let query =

    qObject

    |> Seq.filter(fun*o*-> if storeFilter = "" then true else o.storeCode = storeFilter)

    |> Seq.filter(fun*o*-> if dateFilter = "" then true else o.date = dateFilter)

    |> Seq.filter(fun*o*-> if supplierFilter = "" then true else o.suppName = supplierFilter)

    |> Seq.filter(fun*o*-> if supplierTypeFilter = "" then true else o.suppType = supplierTypeFilter)

# Performance

I carried out multiple tests for different parts of the application. I used the Stopwatch class in both C# and F# to measure the results. For each test I gathered four separate results and excluded any anomalies.

**File Loading**

**C#**

No Parallelism (With loading bar)

1. 25.909 seconds
2. 25.951 seconds
3. 25.742 seconds
4. 25.903 seconds

Parallel (With loading bar)

1. 20.000 seconds
2. 20.496 seconds
3. 20.562 seconds
4. 20.424 seconds

No Parallelism (No loading bar) – *This is what is used in the application*

1. 8.921 seconds
2. 8.744 seconds
3. 8.648 seconds
4. 8.920 seconds

Parallel (No loading bar) – *This caused crashes*

1. 6.388 seconds
2. 6.048 seconds
3. 6.312 seconds
4. 6.148 seconds

The difference between using updated the loading bar each time a file is loaded and without it is quite large. Although the loading bar was a nice addition I did not think it was worth keeping.

**F#**

Parallel

1. 6.930 seconds
2. 6.623 seconds
3. 7.016 seconds
4. 6.715 seconds

The F# parallel speeds are similar to the C# parallel speeds.

**Filtering Orders**

The filters used in each of the following tests were: All stores, All suppliers, Dairy products, All dates.

**C#**

PLINQ

1. 0.429 seconds
2. 0.354 seconds
3. 0.360 seconds
4. 0.359 seconds

LINQ

1. 0.568 seconds
2. 0.543 seconds
3. 0.553 seconds
4. 0.564 seconds

**F#**

PLINQ

1. 0.196 seconds
2. 0.193 seconds
3. 0.203 seconds
4. 0.197 seconds

LINQ

1. 0.194 seconds
2. 0.188 seconds
3. 0.200 seconds
4. 0.312 seconds

Filtering the results is where I saw the biggest difference in speed between the two languages. F# was definitely faster at querying the results than C#.

The average PLINQ query time for C# was 0.375 seconds, whereas for F# the query time took 0.197 seconds on average. I was surprised by how fast regular LINQ is in F# too, as the results between PLINQ and LINQ are very close, and in some cases, LINQ was faster than PLINQ. Whereas in C# there is a clear difference between the LINQ results and the PLINQ results.

# Diagrams

