Design Report for Practical Coursework 1:

Quote Calculator for flooring company

*Aletha Jones - 30047647*

[**Introduction**](#_7ar2opgzuuic)3

[**Design**](#_pvw2g854xhq)4

[Classes](#_8lnnu9k2ar9) 4

[Example](#_c8s9kx70o158) 5

[File](#_6z1fgyfsjowv) 6

[File layout](#_fr6tnq2zwsab) 6

[File reading](#_lswnum8zvitl) 7

[**Implementation**](#_7pyfwfn9pwoo)8

[Archive Class](#_2he0wopcsdf4) 8

[Job Class](#_2tyk0d4d8a8o) 9

[Floor Class](#_nkk16p1apv4) 10

[Shape Structure](#_za3ck3v5rs6o) 11

[Material Structure](#_xe7jquz2kv61) 11

[Customer Class](#_5jfx0zhgr679) 12

[Address Structure](#_yl545627oogb) 12

[**Testing**](#_sfrjnu458zy3)14

[The opening menu](#_mv9290htjyak) 14

[Viewing customers](#_mi8n3w84skag) 14

[Sorting customers](#_q28jqa3vsyjq) 15

[Editing a customer](#_5mupauxl8xfz) 15

[Creating a new customer](#_8wnb6rp03b6l) 17

[Viewing jobs](#_4nbrherpg93a) 17

[Editing a job](#_i6m8vb1e80du) 17

[Creating a new job](#_5fwakexyeewv) 18

[For an existing customer](#_7dbkgyp1yve0) 18

[With a new customer](#_yh2pedbszshz) 19

[Showing different materials](#_g93fgxuh7gd1) 20

[Demo floor](#_4u9ek1sve07o) 21

[Existing floors](#_ne3nah4qgr82) 21

[Exit & Saving](#_i7is3vxn60ud) 22

[**Conclusion**](#_8ptrm49hmrqx)23

[Code Review](#_gdnj3dxze4w9) 24

[**Appendix**](#_igabzizgecl9)26

## Introduction

For this coursework, I chose brief 2: ‘Quote Calculator for flooring company’, and this report will demonstrate how I went about fulfilling this brief and its requirements, detailed in the cover sheet. It will start with the initial design process, followed by reference notes I used to keep track of the objectives, as well as implementation, testing and finally the conclusion in which I share my final thoughts on the project and its components. It’s followed by an appendix of larger pictures of referenced figures.

I chose brief 2 as it was something I hadn’t tried before. Brief 1 (‘Grade Calculator’) was quite similar to a previous tutorial task I had done before. The same goes for brief 3 (‘BMI Calculator’). This was a largely decisive factor, though I also had some ideas related to it as I read through the briefs.

## Design

After reading through the cover sheet, I highlighted the primary data and objects that I’ll be using in the program(Fig 1).

|  |
| --- |
|  |
| *Fig 1* |

### Classes

Using this, I made a preliminary class diagram(Fig 2) to use as a reference in programming. This evolved during the creation of the program(Fig 3).

|  |  |
| --- | --- |
|  |  |
| *Fig 2* | *Fig 3* |
| *Please see Appendix* [*I.A*](#yvuo1vh4sj0y) *and* [*I.B*](#y04yurm84cr3) *for larger images* | |

The inclusion of an extra class *Archive* -to handle the storage of *Jobs* as well as file IO- and another structure *Shapes* came along after releasing that without a dedicated object to handle these, the *main* function would be responsible for handling IO and operations on the stored *Jobs*. I thought a more flexible alternative would be a dedicated ‘storage’ object.

*Shapes* came separately, I wanted to give the user greater flexibility when handling floors. *Shapes* allow the user to make floors beyond rectangles. In a future iteration, I’d like for it to be able to handle a wider range of shapes such as circles and triangles.

#### Example

Figure 4 shows the final rendition of a reference diagram I used while coding and testing. It shows on a basic level the sort of data that each class, structure, array and vector would hold, and how they’d interact with each other. Like the demo file, the data used is based on my OCs from a personal project, serving well for reference and testing.

The diagram also demonstrates the role *Shape* plays in the program. The top floor, belonging to *Jane Derry, is made* up of two shapes, 66m2 and 28m2, to make up a *Floor* area of 94m2. The floor is made up of the *Material*: *wood*, resulting in a total cost of £2,350.

For the second *Customer*, *Daniel Audsworth*, they have two separate *Floors*. One is made up of a single *Shape*, and the other is made of 3. Each *Floor* is made of a separate material and separate groups of *Shapes*.

A flaw with the diagram is that it’s missing the customer’s phone number. I intended on fixing the issue, but the software I use to make notes and diagrams didn’t like how many objects were on the sheet and would repeatedly crash.

|  |
| --- |
|  |
| *Fig 4*  *Please see Appendix* [*I.C*](#w18vpon4x6qq) |

### 

### File

#### File layout

In designing how the program handles reading and writing (R/W) to the file, I put together Figure 5 as a reference for implementation. It’s split between data and size indicators, with each row being a separate line and boxes within a row being separated by commas. I believed it offered a good balance between ease in writing to the file and human readability, though as demonstrated in the next segment, reading from the file was a different story.

|  |
| --- |
|  |
| *Fig 5* |

The file starts by indicating how many *Job*s there are in the file (shown would be 2). It then shows how many lines each job has (per job), followed by the *Job*’s ID. Further on, it lists the *Customer* details (including *Address*es) before listing the floors and their shapes. Following the same idea as before, it lists how many *Floors* there are, then each *Floor* reports how many *Shapes* it has (as well as what *Material* is used).

#### 

#### File reading

During the simultaneous design and test implementations, I had forgotten that the user requires the program to read from the file. This resulted in Figure 6. This shows the process in which the program reads the file, looping back in places for dealing with multiple *Jobs*, *Floors* and *Shapes*.

Figure 7 uses Doxygen to show the calling graph for Archive::ReadJobs

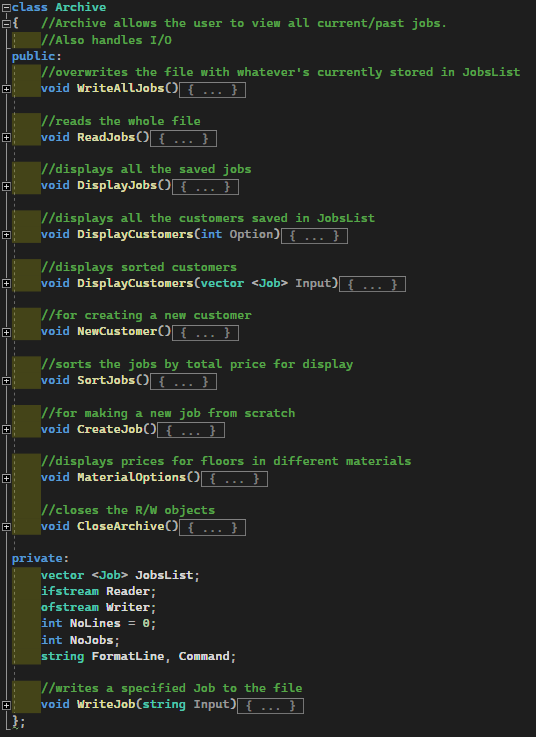
|  |
| --- |
|  |
| *Fig 6* |
|  |
| *Fig 7* |

#### 

## 

## Implementation

### Archive Class



The archive class allows for the *main* function to interact with any stored job. It handles IO with the “floorscustomer.txt” file as well as providing options to sort, create and view jobs and customers.

All the *Job* objects are stored in a vector, allowing for easy additions of new jobs and adding jobs from a file. I also find vectors a lot more versatile and intuitive compared to arrays.

The *WriteJob()* and *WriteAllJobs()* functions are separate as I had intended to allow for a single job to be appended to the end of the file. This wouldn’t work with how the file is set out, so it was left as-is so I could continue with other aspects that required attention.

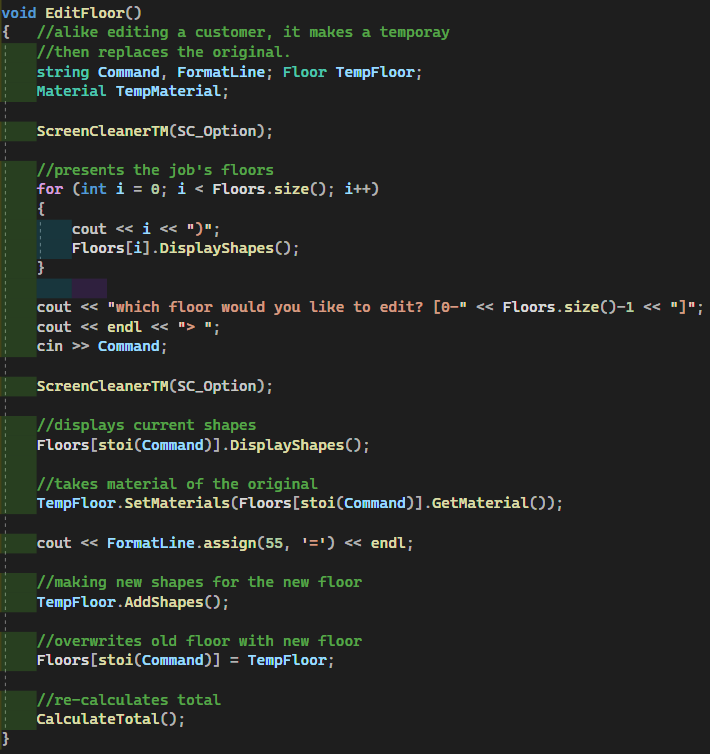
### 

### Job Class

(due to the size of the *JobClass*, it has been placed in the appendix [II.A](#7yr1el5f26ii))

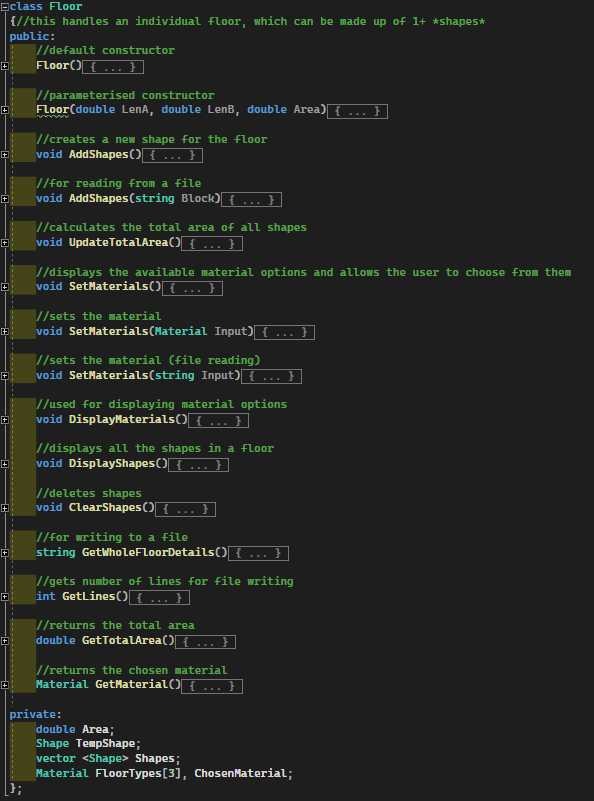
The Job class ties together *Floors* (and *Shapes*) to *Customers* (and *Addresses*), giving *Archive* only one object to focus on and handle. It contains the largest amount of functions, aiming to fulfil most of the requirements set out by the client, from calculating total floor areas to quotes for floor types, adding new *Floors*, creating new *Customers* (per *Job)* and does a large portion of data displaying.

It also handles editing floors and customers (in a similar fashion), though its final form is not as elegant as the original plan. The current version (using *EditFloor()* as the example) displays the current data, then allows the user to make a new *Floor*. This then overrides the original *Floor* data.



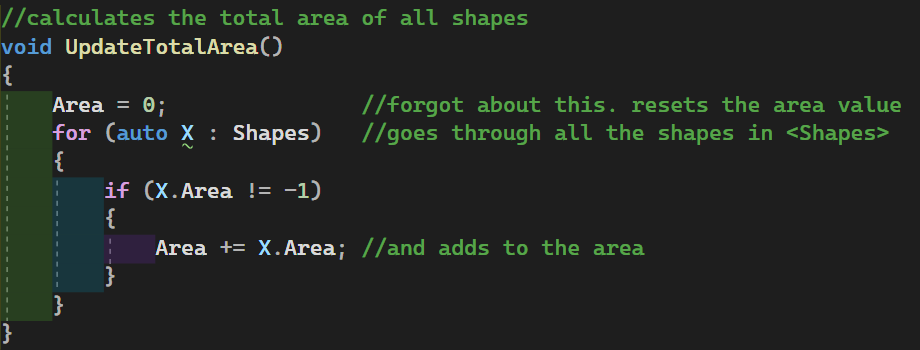
The original design involved a more granular approach, allowing the user to edit individual *Shapes* rather than the whole *Floor* exclusively.

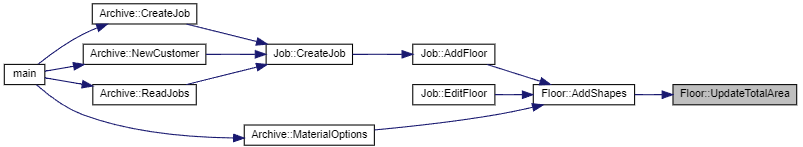
### Floor Class



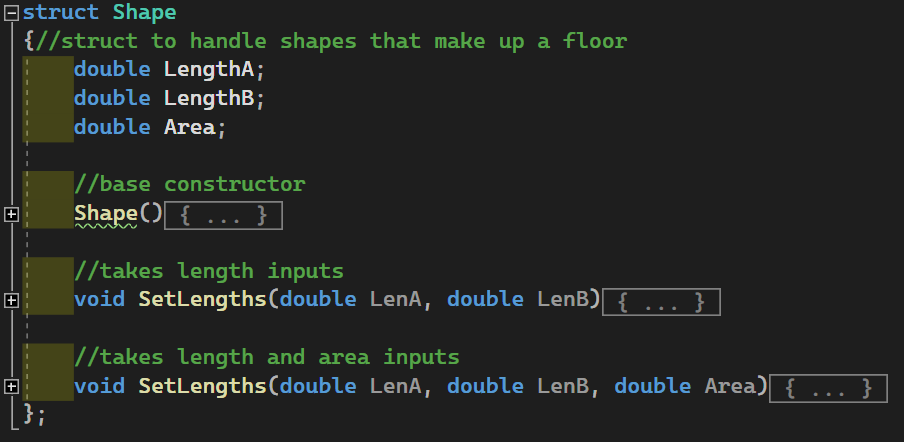
For similar reasons to why *Archive* stores *Jobs* in a vector, *Floor* holds *Shapes* in a vector, allowing for a floor to be composed of a practically infinite amount of shapes. Inevitably, adding any form of limit could result in issues with the user in the field.

Whenever a new *Shape* is added, *UpdateTotalArea()* is called and updates the saved area value.



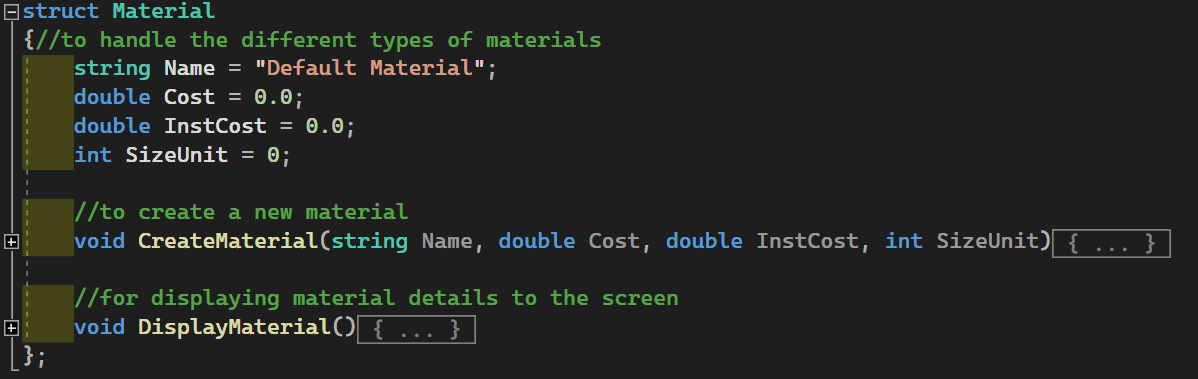


### Shape Structure



The Shape structure handles the basic dimensions (and area) of a shape, along with the ability to set the lengths and area. It’s an important struct that replaces the initial *<Dimensions>* vector of the original design.

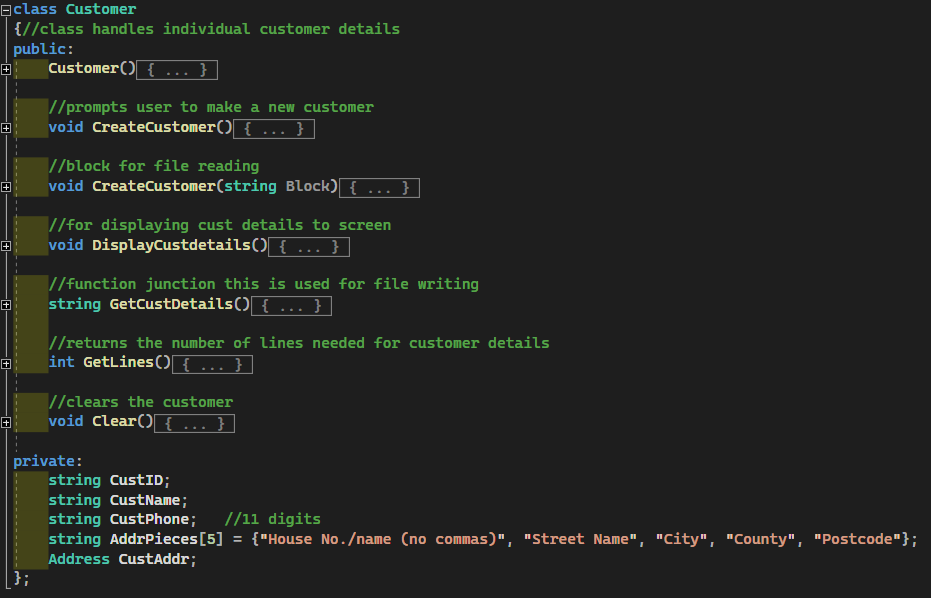
### Material Structure



The Material structure simply handles the details of xyz material. The *SizeUnit* variable determines the size grouping for installation costs. In the brief, it’s given as 5m2:  


Having it as an explicit variable allows it to differ between materials if the client determines it necessary. Though as materials are currently hard-coded (*CreateMaterial()* is future-proofing), it can be expanded upon in future revisions.

### Customer Class

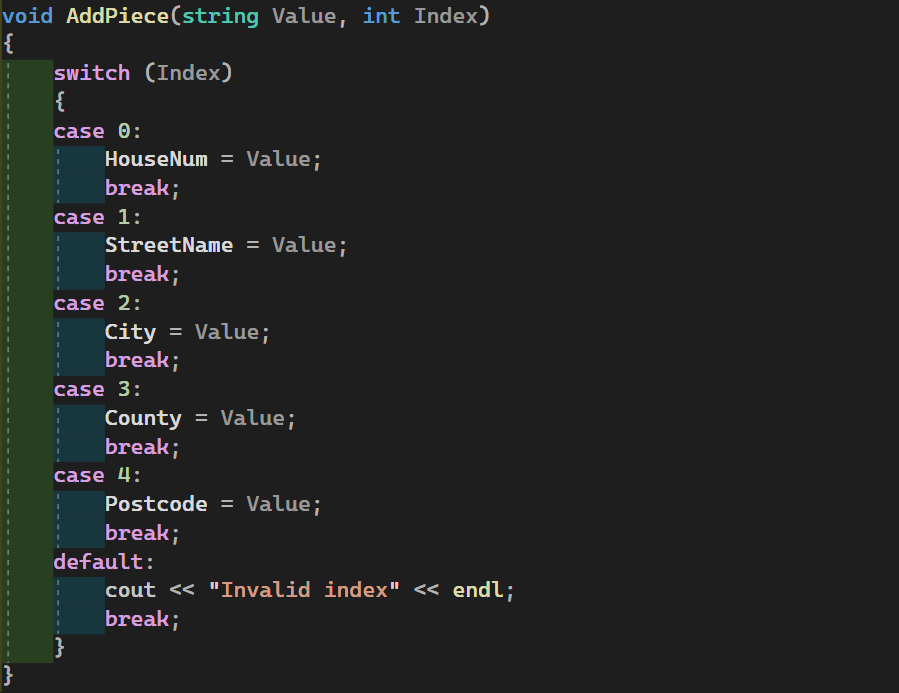


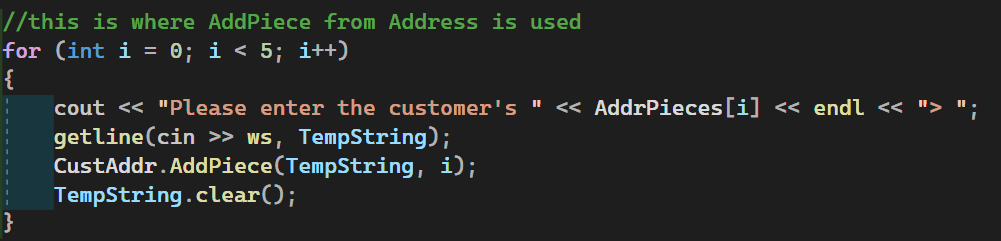
The Customer class handles any data pertaining to the client’s customers. It holds their address, name, ID and phone number as well as provides methods to create a new one and display the current customer. It also holds functions for writing to the file (*GetCustDetails()* & *GetLines()*) and reading (*CreateCustomer(Block)*).

### Address Structure

## 

The Address structure handles the customers address details. It was implemented as a dedicated structure because I thought it would make handling addresses much easier. The *AddPieces()* function is fairly unique compared to most of the functions included in the program. It takes a string value and an integer index.



The function *Customer::CreateCustomer()* goes through a loop, displaying a header for each address line, each corresponding to an index in *AddPiece()*:**

This prevents repetitive lines of cout and getlines and adds another array to the program: (*Customer::AddrPieces[5]*).

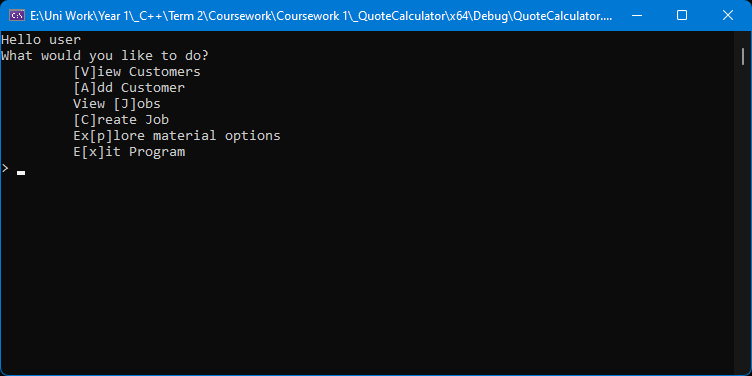


## 

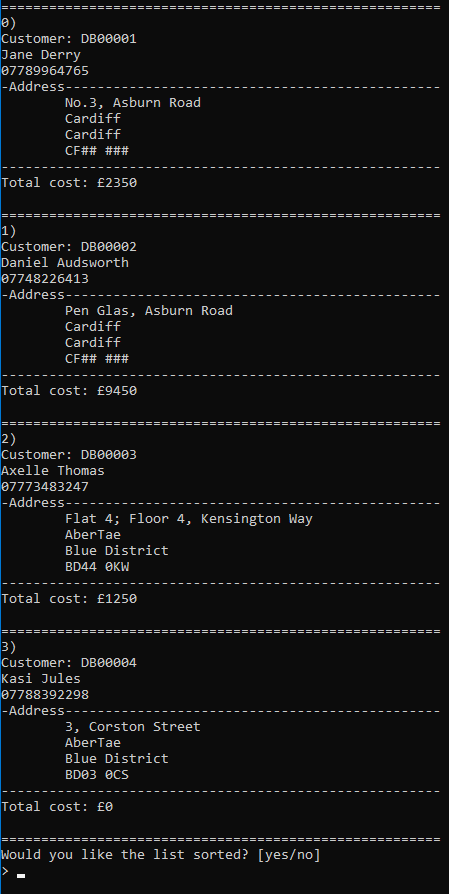
## Testing

When prompted, “return” takes the user back to the ‘opening menu’ and allows them to carry out any other function they’d like.

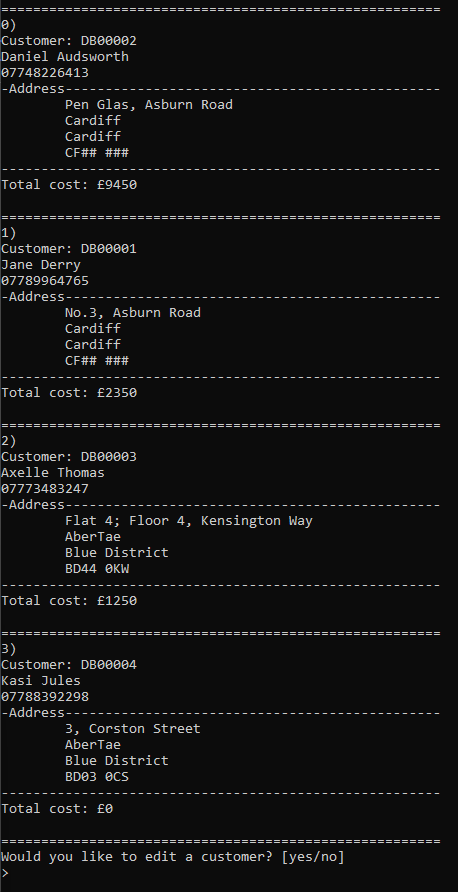
### The opening menu



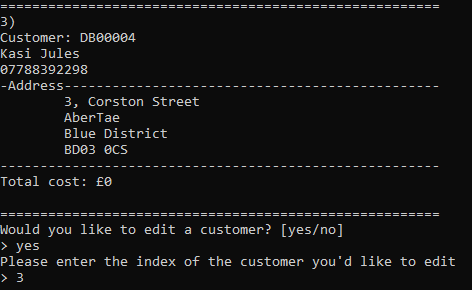
### Viewing customers

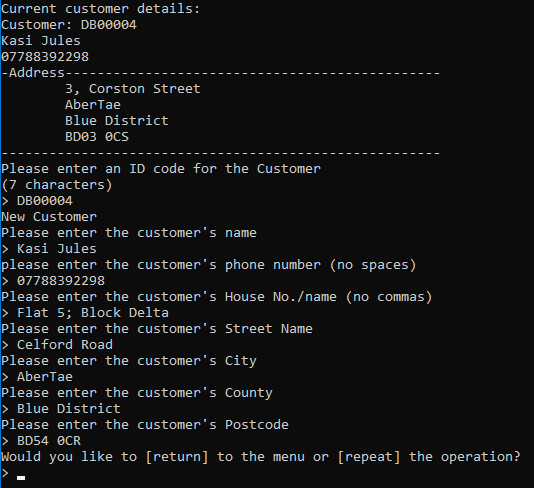


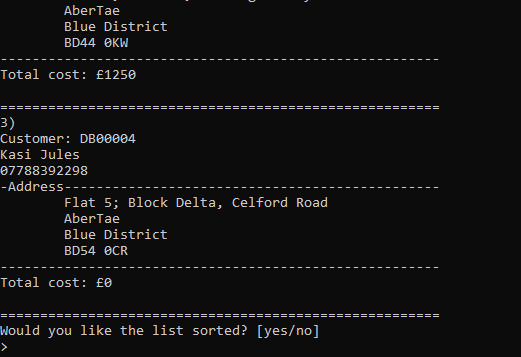
#### Sorting customers



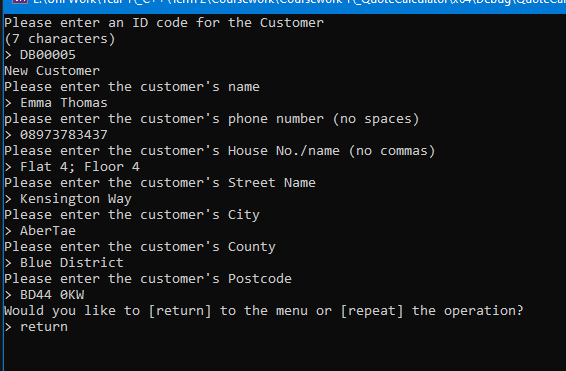
#### Editing a customer

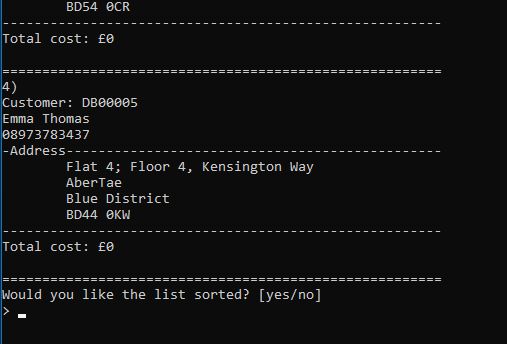






### Creating a new customer

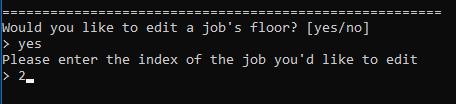




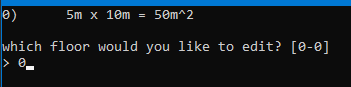
### Viewing jobs

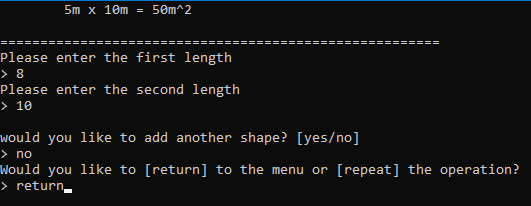
*(Please see appendix* [*III.A*](#eigxjghml71t)*)*

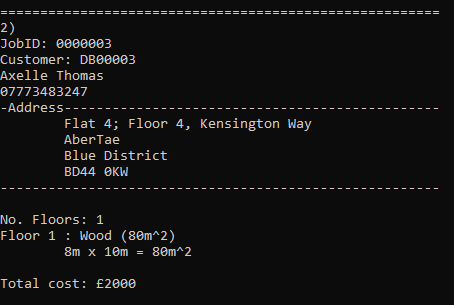
#### Editing a job



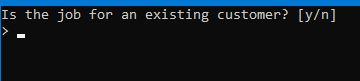
*(Axelle's job)*

**

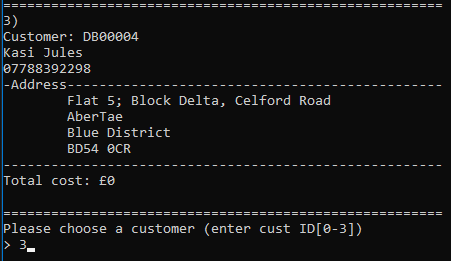
**

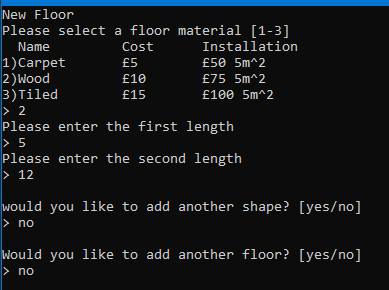
**

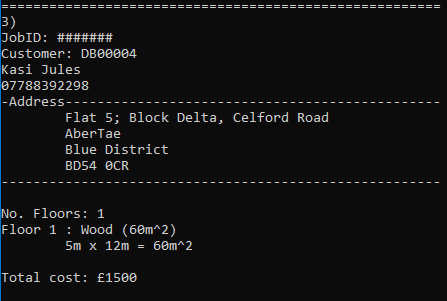
### Creating a new job



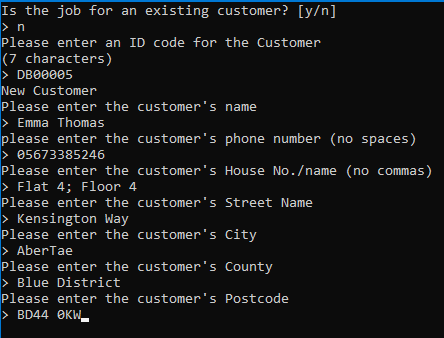
#### For an existing customer

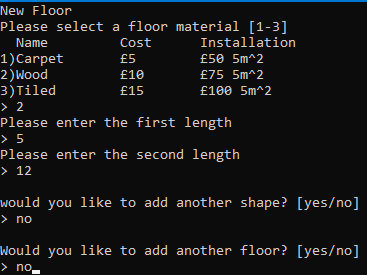




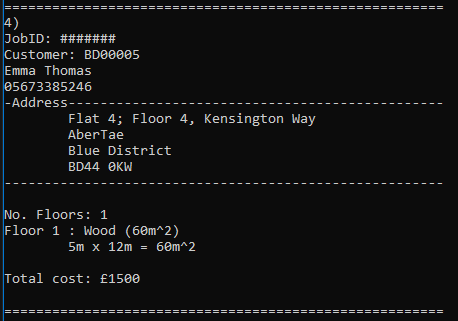


#### With a new customer





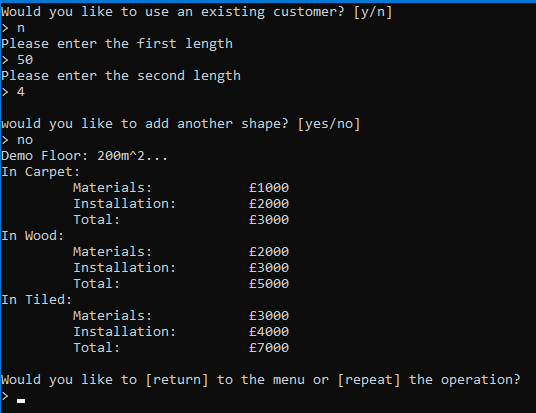
*(viewing jobs)*



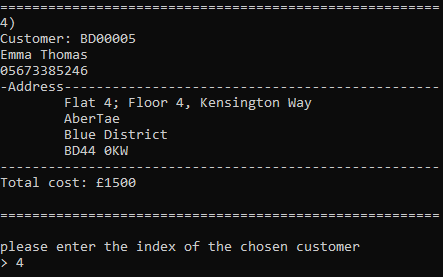
### Showing different materials

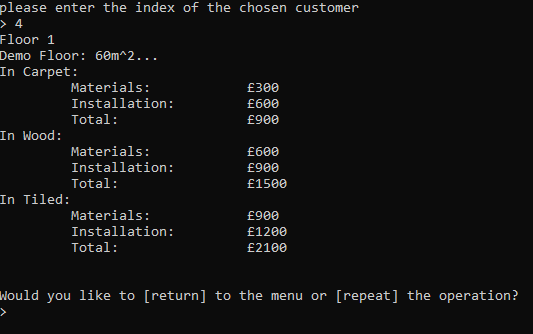


#### Demo floor

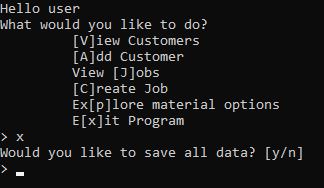


#### Existing floors





### Exit & Saving



## 

## Conclusion

I feel that the program mostly fulfils the client’s spec. On review, I see I misinterpreted “Show customers historic quotes” as showing the user past jobs and customers instead of showing previous quotes given to jobs. That’s something I would fix in a further version. Another thing I’d fix is how the program utilises *CommaRemover* and file reading. As file reading was an unintentional afterthought (and so was *CommaRemover* consequently) its design and implementation weren’t up to standard. It works, just without any elegance. Loosely connected is how the program handles Customer IDs. I would’ve preferred the program to generate an ID, but as it was one of the last functions to be made, I pushed the responsibility onto the user to generate a unique ID.

I’m not entirely pleased with how the menus within the program look, though I’m not entirely sure how I’d improve aside from adding colour. I had intended for the (now ‘hidden’) *Graphics* function to handle colours within the program, but for how much time it consumed attempting to prevent colour spill and consistently get the colours and formatting I was looking for, I dropped the function. It would be a personal priority to work on for a future revision.

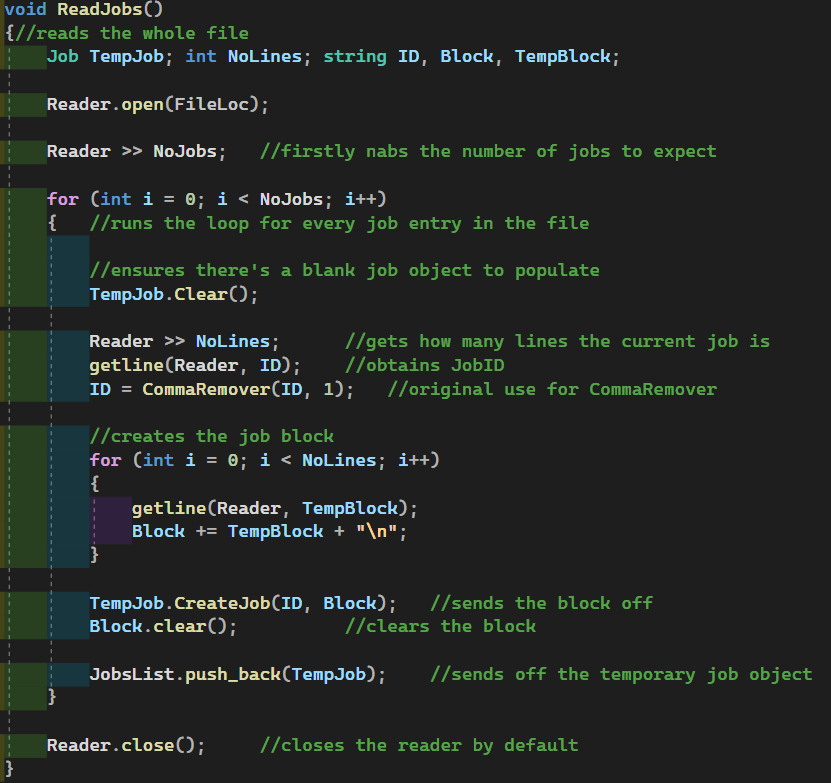
Like how I mentioned at the start of the conclusion, there are some lines of code and bits that are either redundant, unnecessary or I’ve instantly seen how to make something better. A basic example is the *Shape* structure, it doesn’t need two *SetLengths()* as it could have just the first (that takes two lengths) as it calculates the area by default. I can’t think of a specific use case where explicitly inputting a shape’s area is necessary.

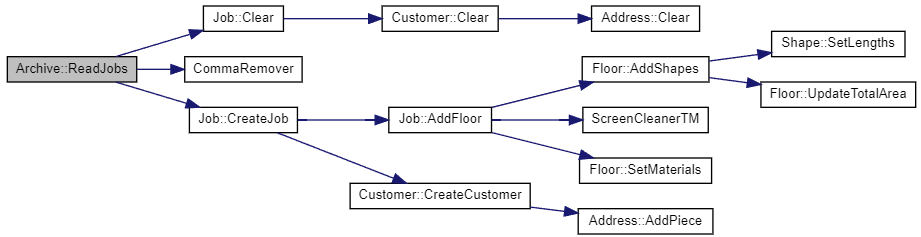
For future projects, to improve my work, I think keeping some form of notes between coding sessions would be useful. Something like a coding journal so that I can see what I added and where I was in the last session, and what I noted I need to fix for the next session. This should greatly reduce overlapping working where I’d make a function I already had without initially realising, and only finding out much later. Though from completing this project, I’ve found zooming in and emboldening the font of the text editor within VS '22 greatly improves productivity. Also disallowing NuGet from “checking for updates” every time I try to build the program reduces how often Visual Studio crashes and/or locks up my PC.

### 

### Code Review

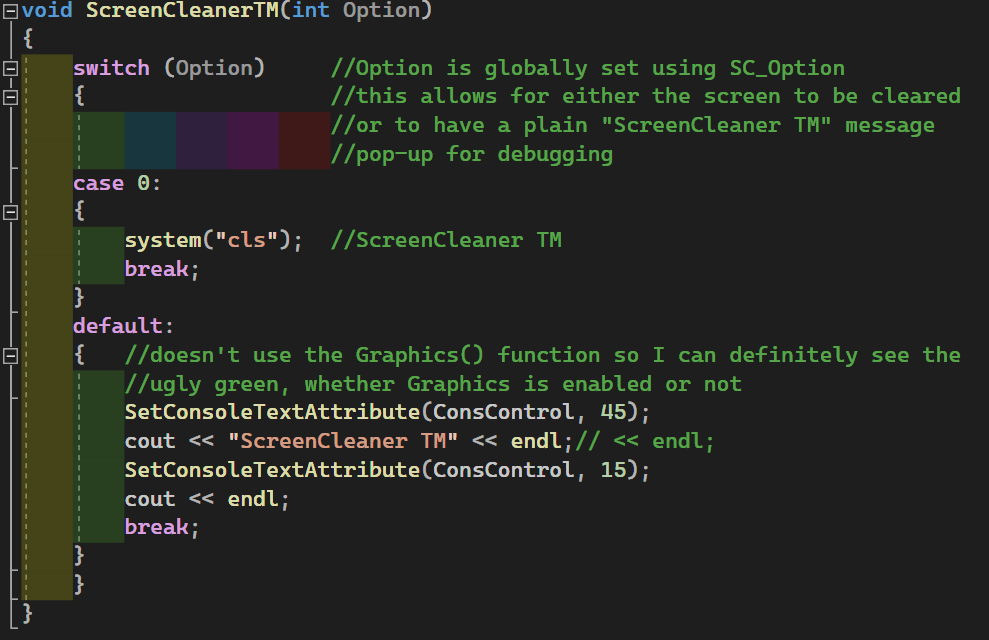
As mentioned above, the file reading is un-elegant in how it operates,



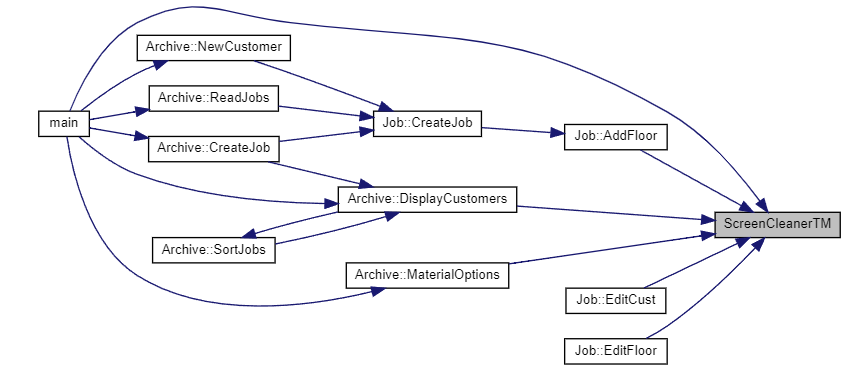
For each *Job*, it makes a ‘block’ which is a single string containing all of a *Job’s* data. This is then passed through to a temporary *Job* object’s *CreateJob()* function. From there it splits up the ‘block’ into a ‘job block’ and ‘customer block’. Using Doxygen, its pathways are shown:

The ‘blocks’ are separated at *Job::CreateJob*, where the ‘job block’ is sent through to *Job::AddFloor* and the ‘customer block’ to *Customer::CreateCustomer*. It works in a similar, reversed way to how the program writes to the file, though it’s written as intended. There wasn’t an intention with writing.

*ScreenCleanerTM()* is a function used throughout the program to clear the contents of the screen.



Using Doxygen, its call diagram is shown below.

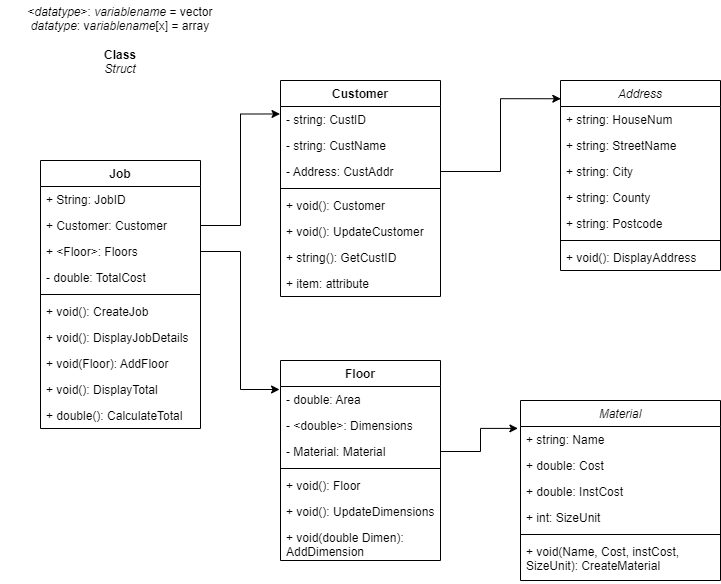
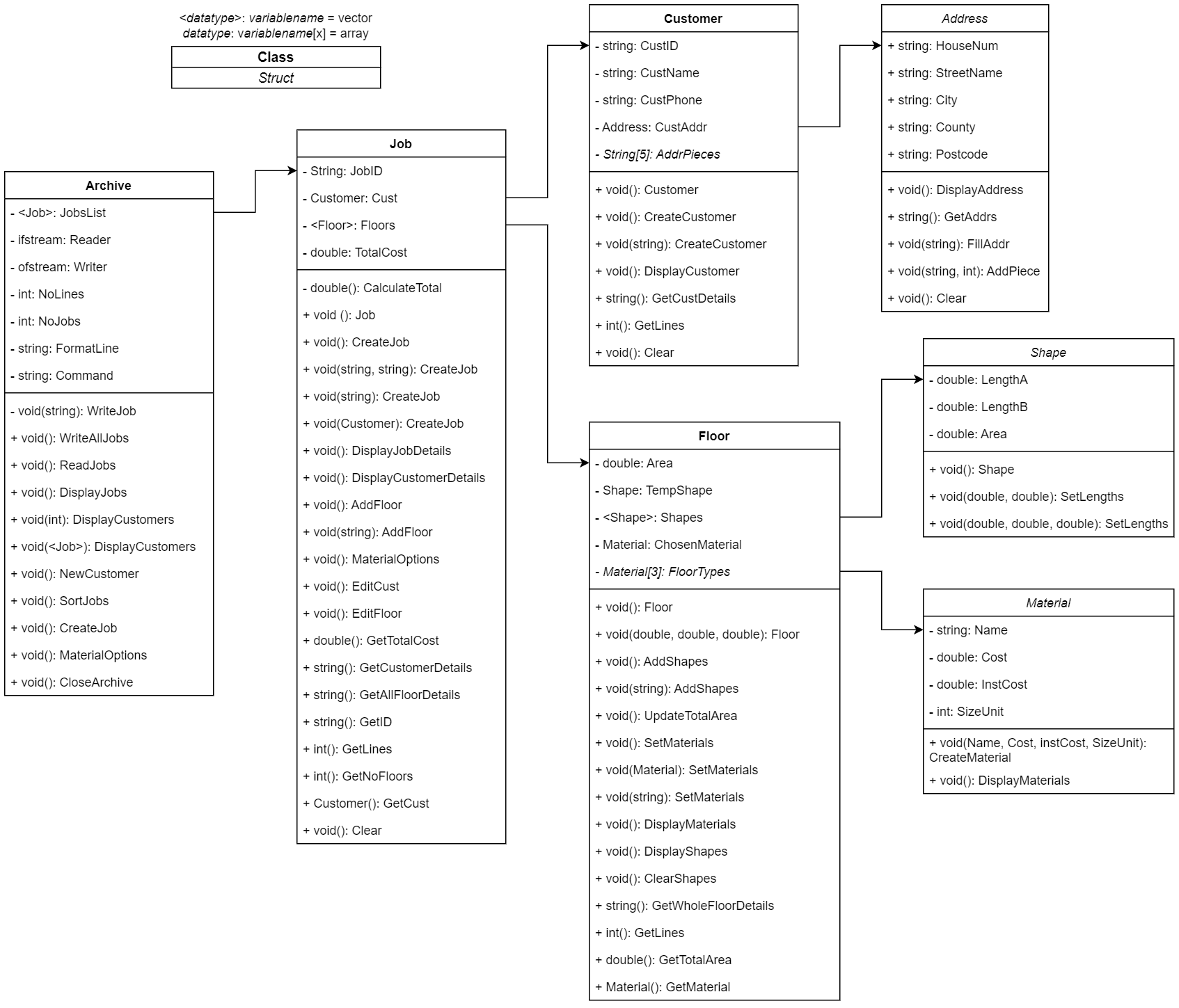
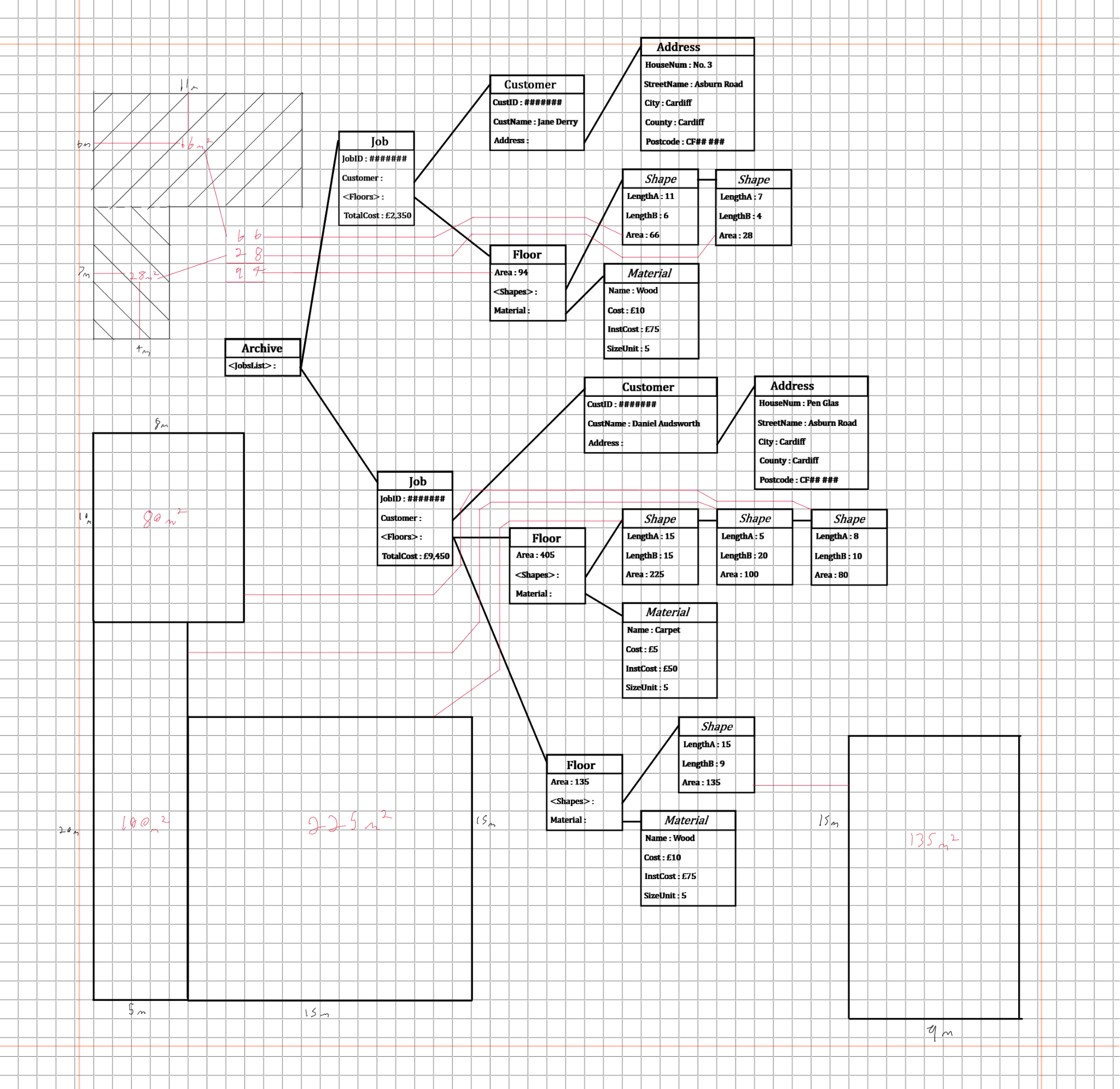
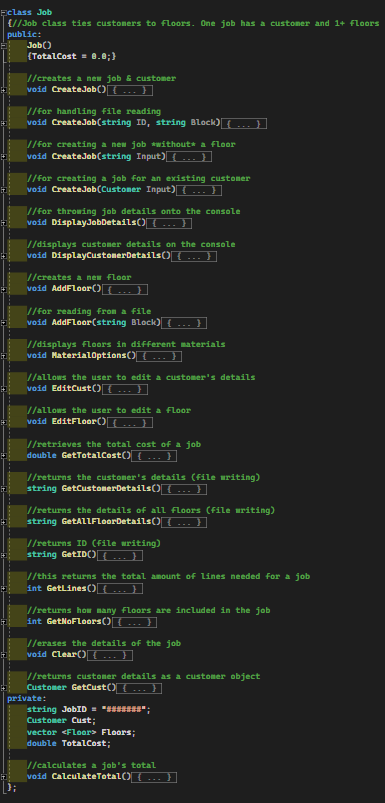


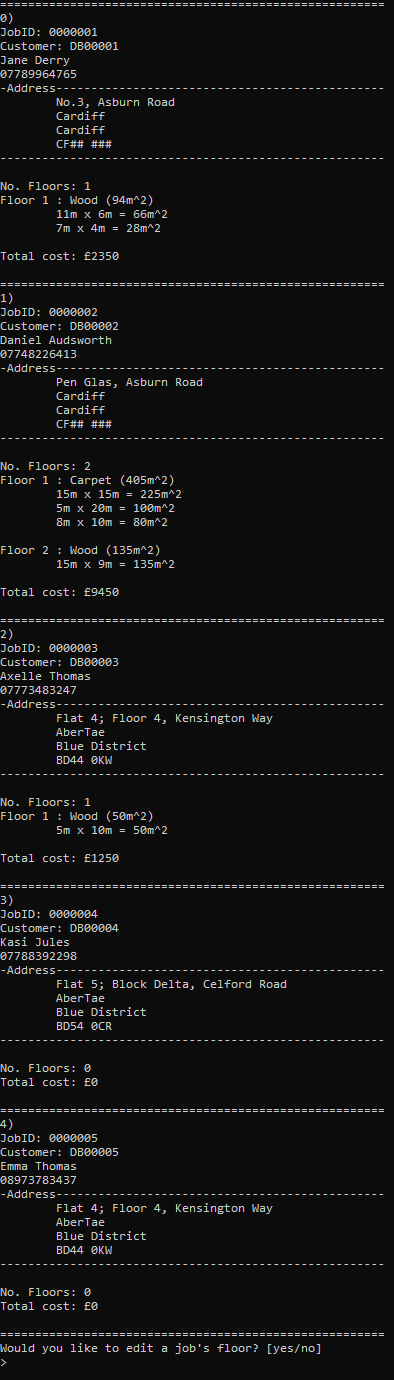
Out of the project, I’m most satisfied with how *ScreenCleanerTM()* works. At the top of the file, a global integer variable *SC\_Option* is declared, and it determines which *case* of *ScreenCleanerTM()* it should use. The options are largely based around debugging, setting *SC\_Option* to 0 clears all text from the screen. Setting it to 1 instead displays



…on the console. This keeps data viewable on screen to keep track of where the program goes. It’s set to 0 by default.

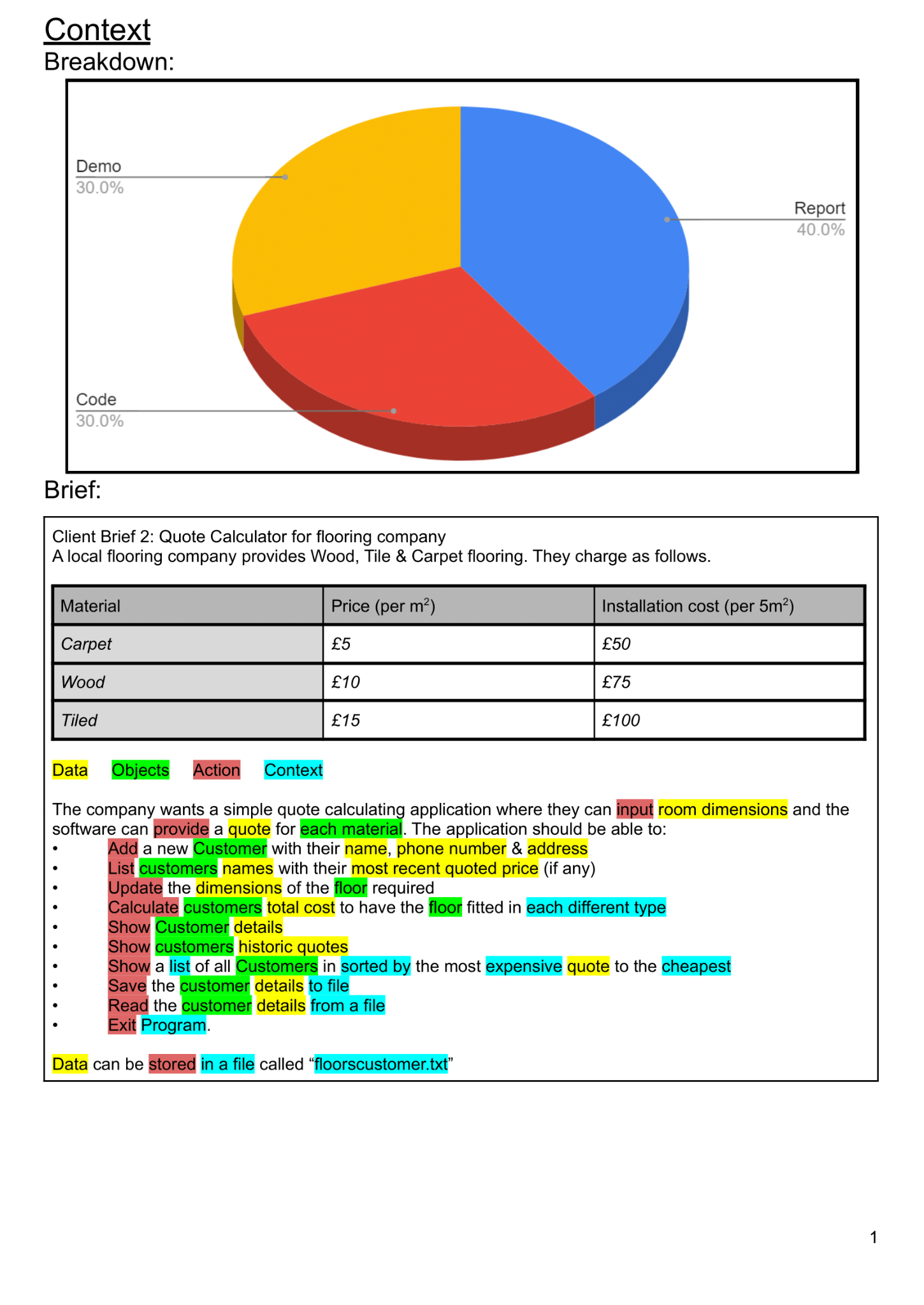
## Appendix

* 1. Original class diagram ([Jump Back](#kbx06qqkjr42))  
       
     
  2. Final class diagram (Jump Back)  
     
  3. Reference Diagram ([Jump Back](#hkvfrpcl9o0v))  
     
  4. *JobClass* ([Jump Back](#xsgnnxgjf1t4))  
       
     

1. .
   1. View Jobs ([Jump Back](#a1igsokgp9b6))

X.

1. Initial reference notes (main)



B. Testing file

